Software Engineering Institute

The Evolution of a Science Project: A Preliminary System Dynamics Model of a Recurring Software-Reliant Acquisition Behavior

William E. Novak
Andrew P. Moore
Christopher Alberts

July 2012

TECHNICAL REPORT
CMU/SEI-2012-TR-001
ESC-TR-2012-001

Acquisition Support Program

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by United States Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the

Contracting Officer
ESC/CAA
20 Shilling Circle
Building 1305, 3rd Floor
Hanscom AFB, MA 01731-2125

Table of Contents

Acknowledgments **v**

Executive Summary **vii**

Abstract **ix**

1 Introduction **1**
- 1.1 Objectives 2
- 1.2 Approach 3
- 1.3 Assumptions and Constraints 3

2 Prior Research in Modeling Rework **5**

3 Acquisition Dynamic: The Evolution of a Science Project **6**
- 3.1 Concept Criteria and Selection 6
- 3.2 Acquisition Program Example Narrative 7
- 3.3 Qualitative Analysis of the Dynamic 12
  - 3.3.1 Misaligned Incentives and Structural Dynamics 12
  - 3.3.2 Incentives Driving “The Evolution of a Science Project” 13
  - 3.3.3 User Demand, Satisfaction, and Schedule Pressure 13
  - 3.3.4 User Community Management 14
  - 3.3.5 Uncontrolled Prototype Growth 14
  - 3.3.6 Project Manager Decisions 15
  - 3.3.7 The 90% Syndrome 15
  - 3.3.8 Re-Architecting the System 16
  - 3.3.9 Project Management Infrastructure 16
  - 3.3.10 The Transition from Science Project to Formal Development Program 16
  - 3.3.11 Evolutionary Prototyping vs. Throwaway Prototyping 17

4 System Dynamics Model and Simulation Results **19**
- 4.1 Dynamic Hypothesis 19
- 4.2 Simulation Model 23
  - 4.2.1 Science Project Sector 27
  - 4.2.2 Production Development Sector 28
- 4.3 Preliminary Simulation Results 30

5 System Dynamics Modeling Lessons Learned **39**
- 5.1 Tooling 39
- 5.2 Modeling Process 39
- 5.3 Using the Model 41

6 Summary, Conclusions, and Next Steps **43**

7 References/Bibliography **48**

Appendix A: System Dynamics Method Background **51**
- 7.1 Cooper 51
- 7.2 Abdel-Hamid and Madnick 51
- 7.3 Ford and Sterman 53
- 7.4 Taylor and Ford 53
- 7.5 Madachy 54

Appendix B: System Dynamics Method Background **56**
7.6 Recommended Resources for System Dynamics Modeling 60

Appendix C: System Dynamics Simulation Model: Science Project Sector **62**

Appendix D: System Dynamics Simulation Model: Production Development Sector **63**

Appendix E: Interface for Interacting with the Model **64**

Appendix F: Simulation – Science Project Dynamics **66**

Appendix G: Simulation - Applying Pressure to Workers **69**

Appendix H: Simulation – Scoping the Science Project Effort **72**

Appendix I: Simulation – Decision to Reuse the Prototype **74**

Appendix J: Potential Enhancements to the Model **76**

List of Figures

Figure 1: Overview of Model Development Process 19

Figure 2: Notional Problem: Ripple Effect Due to Incremental Discovery of Science Project Quality Issues 20

Figure 3: 90% Syndrome Due to Rippling Rework in the Production Development 21

Figure 4: Causal Loop Diagram of "The Evolution of a Science Project" 22

Figure 5: Overview of the System Dynamics Model 26

Figure 6: Stocks and Flows in the Science Project Sector 27

Figure 7: Stocks and Flows in the Production Development Sector 29

Figure 8: Simulation Results for Production Development Discovered Quality Issues 31

Figure 9: Simulation Results for Science Project Rework to be Discovered 32

Figure 10: Simulation Results for Percentage Complete 33

Figure 11: Simulation Results for Production Development Discovered Quality Issues 35

Figure 12: Simulation Results for Production Development Scheduled Completion Date Reflecting the Decision to Reuse 36

Figure 13: Simulation Results for Production Development Discovered Quality Issues Reflecting the Decision to Reuse 36

Figure 14: Simulation Results for Production Development Scheduled Completion Date for Various Worker Pressures 37

Figure 15: Stock and Flow Diagram of Rework Model [Ford 1998] 53

Figure 16: Causal Loop Diagram of Rework Model [Taylor 2005] 54

Figure 17: A Simple Feedback Loop 58

Figure 18: (a) Project Management—Desire to Use Overtime to Correct Schedule; (b) Closed-Loop Representation Showing (Balancing) Feedback to Improve Progress 59

Figure 19: Unintended Burnout Due to Overtime 60

Acknowledgments

Many people have worked to sponsor and improve this report and the work it describes, and it would not have been possible without their support, help, and expertise. First and foremost, we would like to thank our sponsor, the office of the United States Assistant Secretary of Defense (Research and Engineering) for the funding and opportunity to perform this work. We would also like to thank the SEI's Acquisition Support Program Executive Director Terry Roberts and Deputy Director Mary Catherine Ward for their support and encouragement. Finally, the authors owe a debt of gratitude to the people within the SEI who provided technical review and comment on this report, including John Foreman, Julie Cohen, and Jay Marchetti, as well as our editor, Gerald Miller. Due to their efforts and expertise, this is a much better document.

Executive Summary

A recurring theme in the Software Engineering Institute's assessments of acquisition programs is the prevalence of "science projects"—programs that start as small, informal prototypes to show proof of concept, and are then forced to grow into full production systems to meet demand, but find it difficult to do so. This dynamic is called "The Evolution of a Science Project," and this report describes the findings of the initial year of work which has been spent analyzing the behavior using system dynamics modeling and simulation.

The key preliminary findings from simulation runs of the system dynamics model are as follows:

1. The accumulation of undiscovered rework from the science project creates a "tipping point"[1] during production development (*at the tipping point, program behavior changes unexpectedly from completing in a finite time, to not completing at all*).
2. Applying high pressure to developers, or even moderate pressure for extended periods, can push the program past the tipping point.
3. The tipping point contributes to the "90% Syndrome"[2] (*this occurs when development stalls near completion, with little progress made despite spending more time and effort*).
4. Applying only a low level of pressure to workers for a limited time, even in the face of heavy schedule demands, shortens project duration through increased productivity.
5. From a technical standpoint, an earlier transition from the science project to production development creates fewer problems than does a later transition.
6. The project performs significantly better by shelving the prototype when it is complete, instead of reusing a poorly-constructed prototype.

The high-level conclusions resulting from this research include the following:

- Many software-reliant acquisition program behaviors are an interaction between the inherent structural dynamics of the software development activities and the incentives that are acting on the key participants and stakeholders as they make decisions.
- System dynamics modeling can provide a deep understanding of the often poorly understood dynamics of software-reliant acquisition.
- It's helpful to have a clear understanding of a given dynamic before selecting the most appropriate course of action for a given program (so that the action addresses the actual problem, rather than only a symptom).
- Tipping points may play a key role in many complex and challenging software development and acquisition situations, and require further study.
- Building models that are valid and reliable is a significant undertaking, requiring significant effort, as well as both detailed domain expertise and extensive modeling experience.

[1] A tipping point is a threshold that, when crossed, causes a significant change in the output, despite only an incremental change in the input.

[2] The 90% Syndrome refers to a point at which development appears to be largely complete, but measurements of forward progress stall while the remaining work seems to continue to expand.

- There is much more that can be learned about the problematic behaviors of software acquisition programs and their potential solutions by applying system dynamics modeling to other aspects of software acquisition.

The report also provides a set of “lessons learned” regarding the use of system dynamics modeling.

Modeling incentives and dynamic behavior of acquisition programs is a powerful and cost-effective way to understand problems, evaluate candidate solutions, and produce better program outcomes. Even the most minor efficiencies gained in large acquisition programs from using these methods would produce benefits in functionality, quality, performance, improved time-to-deployment, and direct cost savings that would recoup the needed research investment many times over.

Abstract

Analysis work by the SEI on data collected from more than 100 Independent Technical Assessments (ITAs) of software-reliant acquisition programs has produced insights into some of the most common ways that programs encounter difficulties.

This report describes work done at the SEI that is based on these insights, and intends to mitigate the effects of both misaligned acquisition program organizational incentives, and adverse software-reliant acquisition structural dynamics, by improving acquisition staff decision-making. The research presented here uses a preliminary system dynamics model to analyze a specific adverse acquisition dynamic concerning the poorly controlled evolution of small prototype efforts into full-scale systems that is called "The Evolution of a Science Project."

The report provides a narrative from an actual acquisition program that exemplifies the dynamic, qualitatively describes its key aspects, and presents some of the most relevant prior research done on one of those aspects, project rework. The system dynamics model of the behavior is described in detail, along with the process by which it was developed, and the results of simulations run using the model. The report concludes with a set of lessons learned about system dynamics modeling, as well as potential future research directions.

1 Introduction

Software-reliant systems are those that rely on software to provide their core mission functions—and the failure of programs developing these systems is an all-too-common occurrence for modern government acquisition. The SEI is periodically asked to investigate the reasons why some software-reliant acquisition programs are seriously challenged, and even fail, despite the best efforts by government and industry to make them successful. The SEI works closely with, and in many cases conducts assessments of individual software-reliant acquisition programs that are facing such challenges. These engagements have provided visibility into the processes and forces at work within these programs. Analysis work by the SEI on data collected from over 100 Independent Technical Assessments (ITAs) of software-reliant acquisition programs has produced insights into some of the most common ways that programs encounter difficulties. At a high level it has become evident that acquisition programs continue to regularly experience recurring cost, schedule, and quality failures, and program progress and outcomes often appear to be unpredictable and unmanageable [GAO 2006]. Furthermore, many acquisition leaders and staffers neither recognize these issues (despite their regular recurrence), nor understand how to manage them [Kadish 2006]. Despite best efforts by the Department of Defense to train and equip the acquisition workforce with the skills and guidance it needs to manage large, complex programs, significant challenges remain. Government and industry have an urgent need to help the defense acquisition workforce and their defense industry counterparts better recognize and understand the reasons behind the failures of many acquisition programs, and to provide guidance on how to mitigate or avoid these problems in the pursuit of better outcomes.

One growing theme in acquisition is the prevalence of programs that are sometimes called "science projects," (which may take the form of rapid prototyping efforts, advanced research projects, or Advanced Concept Technology Demonstrations) and the difficulties associated with evolving such efforts into larger and more formal acquisition programs. The name "science project" refers to a program that starts out small and informally, often as an experimental development or prototype system to show proof of concept, and then grows in scope as demand increases. This report describes a commonly recurring pattern of behavior seen by the SEI in software-reliant acquisition programs called "The Evolution of a Science Project," and its analysis using system dynamics modeling and simulation. This pattern was identified over the course of multiple ITAs that gathered qualitative data on the behavior, and was endorsed as being timely and relevant in the current acquisition environment by both acquisition practitioners and executives.

This report represents the results developed in the initial year of work devoted to this effort. While an extensive preliminary model was completed which provides useful and significant insights and conclusions, some planned additional extensions and enhancements to the model could not be implemented in the first year of work (see Appendix J for a list of these areas). We plan to continue this effort to further develop and analyze the model in subsequent work.

The report is structured as follows:

- Section 1 presents the objectives, approach, and assumptions and constraints of this work.
- Section 2 discusses five instances of prior research in modeling rework, a key underlying aspect of the "Evolution of a Science Project" dynamic.
- Section 3 describes the dynamic in detail at a qualitative level, along with the key issues that drive it.
- Section 4 describes the system dynamics model that was developed, and the preliminary results that were obtained from running simulations.
- Section 5 describes a set of lessons learned over the course of this work on performing system dynamics modeling of software-reliant acquisition.
- Section 6 presents the key conclusions that were derived from this work, and proposes possible next steps for its continuation.

1.1 Objectives

Our purpose in the work described here has been to a) model and understand a timely and relevant dynamic that recurs frequently in software acquisition programs, and b) create a foundation of expertise and system dynamics modeling infrastructure that can be built upon in subsequent work.

The key questions that this work is attempting to answer are as follows:

- Can the management and software development aspects of software-reliant acquisition programs be cost-effectively modeled using system dynamics such that the simulation results reasonably and consistently conform to the outcomes seen in actual programs?
- Can the results of such modeling be used to produce practical insights into different ways to improve the conduct and management of software-reliant acquisition programs?

There are, of course, additional questions that could, and should, be answered as well, but which could not be included within the limited scope of this initial effort, such as:

- Can candidate solutions and mitigations for the counter-productive dynamics exhibited by the model be accurately modeled, and their potential viability evaluated using system dynamics techniques?
- Can the models and the insights gained from these models be used to help the acquisition workforce significantly improve the quality of their decision-making regarding counter-productive dynamics, producing better outcomes for acquisition programs?

The intended audience for this report is twofold: software-reliant acquisition program office leaders and acquisition contractors, as well as those studying software-reliant acquisition and development, or system dynamics modeling. It is valuable to program management office (PMO) leaders and contractors because the descriptions, observations, and conclusions presented here on "The Evolution of a Science Project" dynamic are relevant and applicable to the successful completion of any acquisition program that intends to follow a similar approach. PMO staff or others with project management experience may wish to focus primarily on Section 3, as well as the re-

sults presented at the end of Section 4, System Dynamics Model and Simulation Results. The report may be useful to those interested in acquisition or modeling due to its description of applying system dynamics modeling to the study of software-reliant acquisition as a complex, dynamic system it its own right. For those reading the full report, a working knowledge of project and program management, as well as a basic understanding of system dynamics (especially for sections 2 and 4) will be useful in understanding some of the concepts. For those interested in gaining a better understanding of the methods applied here, Appendix B provides a brief background on the system dynamics approach.

1.2 Approach

System dynamics is an increasingly widely used approach to modeling and simulating the behavior of complex dynamic systems. System dynamics has been successfully used to model project behavior of many different kinds ([Sterman 2000, Repenning 2001, Taylor 2005], and others), including software development projects ([Abdel-Hamid 1991, Raffo 1999, Scacchi 1993, Madachy 2006]), and to a lesser extent, acquisition ([Cooper 1980, Brougham 1999, Haberlein 2004, Adams 2004]) and even software-reliant acquisition programs ([Scacchi 2001, Buettner 2008, Choi 2006]).

The basic steps of the system dynamics modeling process were applied in the development and analysis of the "Evolution of a Science Project" model. The process that was used consists of six steps:

1. define problem
2. formulate/refine dynamic hypothesis
3. formulate/refine system dynamics model of problematic behavior
4. determine if model exhibits the right behavior for the right reasons
5. propose and evaluate problem mitigations
6. revise organizational policies/decision-making and declare modeling effort complete

Section 4, System Dynamics Model and Simulation Results, details the application of these steps to the development of the model.

1.3 Assumptions and Constraints

The software-reliant acquisition dynamic described in this report, while based on actual software acquisition programs, is intended to describe a category or type of program—an amalgam of multiple different, but related programs. Even so, the details of program behavior that are required by the mathematics behind system dynamics modeling should be supported with specific quantities, precise equations, and well-defined relationships. Unfortunately, all too often acquisition programs do not track metrics for many of the program attributes that are needed by a model. In fact, some of these attributes are not even formally recognized, much less as being significant—although their consequences can be very real. Some examples of such attributes that may become variables in a system dynamics model of software acquisition programs include *user demand*, *user satisfaction*, *schedule pressure*, *sponsor confidence*, and many others. Sponsor confidence, for example, is rarely (if ever) measured, but is acutely felt by members of the program staff, and its

continued decline can have catastrophic effects on the program's outcome. None of these variables has a precise definition, nor associated units of measurement, and as a result, definitions must be invented by the modelers, values must be estimated, and the behavior of the attribute must be inferred from the experience of domain experts. In the eyes of some this relegates system dynamics modeling to the realm of "soft science," making its usefulness suspect. The reality, however, is that the lack of reliable, quantitative data on acquisition program performance extends well beyond the definition of "soft" variables, and into even fundamental aspects of acquisition program behavior: productivity, duration, cost, and quality. While most programs claim to measure their cost, performance, and schedule, this measurement is done inconsistently across programs with different definitions for these concepts, and in many cases using different units of measure (i.e., KSLOC/person-month vs. function points/person-month, etc.). In some cases even the underlying measure of effort expended (i.e., hours charged) isn't consistent because it is based on the number of hours *charged*, which does not always accurately reflect the number of hours *worked*. So while the precise values of soft variables may be called into question, in practice the presumably "hard" variables from empirical data collection on acquisition programs are also suspect.

2 Prior Research in Modeling Rework

The concept of rework, which is a central idea behind the dynamics of "The Evolution of a Science Project," is a major part of most project development models, and has received considerable attention in the research literature. Rework refers to the production of work in which some portion of the work has been done incorrectly, and is then classified as either discovered or undiscovered (i.e., latent) rework which will need to be performed again, often requiring unplanned effort. Rework is also sometimes referred to as "firefighting," in the sense that as defects are inserted into the system, developers may be diverted to fix them, thus slowing progress and increasing schedule pressure, which can shortcut quality assurance processes, and so inject more defects.

Perhaps the most interesting aspect of rework is that rather than having been studied and completed by the research community, rework continues to be the subject of new research in system dynamics [Lyneis 2007, Ford 2007, Li 2011a, Li 2011b]. This is perhaps because rework is inherent in software development, often resulting in substantial cost and schedule growth, in part because of the feedback that it presents. Without tooling such as system dynamics modeling that can be used in its study, rework presents a complex and nonlinear problem that is generally intractable to traditional analysis methods.

For additional information about how rework has been addressed previously, Appendix A contains descriptions of the approaches used to model rework in five seminal system dynamics modeling efforts:

- Ken Cooper, "Naval Ship Production: A Claim Settled and a Framework Built" [Cooper 1980]
- Tarek Abdel-Hamid and Stuart Madnick, *Software Project Dynamics* [Abdel-Hamid 1991]
- David Ford and John Sterman, "Dynamic Modeling of Product Development Processes" [Ford 1998]
- Tim Taylor and David Ford, "Why Good Projects Go Bad: Managing Development Projects Near Tipping Points" [Taylor 2005]
- Raymond Madachy, *Software Process Dynamics* [Madachy 2008]

3 Acquisition Dynamic: The Evolution of a Science Project

The recurring dynamic of science projects has received relatively little discussion in the acquisition literature, despite the fact that the pattern has become increasingly common. Due in part to urgent demands for new systems and technologies in theaters such as Iraq and Afghanistan, a significant number of the challenged programs the SEI assesses with ITAs could be characterized as having begun as science projects.

Many defense programs that started out as science projects ultimately produced important advances in technology that leapfrogged our adversaries and gave American warfighters a critical edge in conflicts around the world. In many cases the ability to quickly develop and deploy a new technology has been key to adapting effectively to rapidly changing conditions and threats. The question is not whether we *need* science projects, but how we can most effectively—and sustainably—move the technology from a prototype into the hands of the user or warfighter.

The "Evolution of a Science Project" dynamic describes a program that begins as a small research effort to build a prototype system to produce a compelling proof of concept to help solicit funding. If the system is successful in a test deployment to users, it creates demand for the capability, and the user community may insist that it be developed further and deployed more widely. This pressing need for the capability can drive the prototype system to be deployed, and force the program down a path of rapid incremental development. Since neither the software itself, nor the original project management has the design or robustness needed for production deployment, significant problems soon occur. The project may try to develop a full-scale production system on top of a prototype, with the resulting system exhibiting issues with robustness, capability, performance, and usability. Also, the project's infrastructure (staff size, processes, experience, etc.) that may have been adequate for a prototype are now inadequate for a formal acquisition program. Thus, the project may have great difficulty in successfully making the transition from being an informal project to becoming a rigorous formal acquisition program. In an environment of scarce funding and chronic schedule pressure, there is substantial temptation to maintain the original project infrastructure, and deploy and continue to incrementally develop on top of the prototype base.

3.1 Concept Criteria and Selection

This dynamic was chosen to be the basis of this effort because it met a number of important criteria that help to ensure the dynamic will be relevant to software-reliant acquisition practitioners:

- *Software Relevance*: It is relevant to software development and software engineering because it displays properties that are unique to those domains.
- *Acquisition Relevance*: It exhibits key aspects of acquisition management that directly affect the program's outcome.
- *Acquisition Program Example*: There must be access to information and data on one or more known instances of the dynamics occurring in actual acquisition programs. The dynamic must be based on actual program behavior, as opposed to being only a theoretical possibility.

- *Counter-Intuitive Aspect*: The dynamic should produce an outcome for the program which is either counter-intuitive, or presents counter-intuitive aspects (i.e., it is not intuitively obvious that the dynamic will play out as it does). Otherwise, the dynamic is unlikely to present a significant challenge to program managers.
- *Richness*: The dynamic must present sufficient complexity that the dynamic is worthy of modeling and analysis. Dynamics that possess multiple linked feedback loops, or even exhibit multiple dynamics simultaneously, are most likely to exhibit complex, nonlinear behaviors that create management difficulties.
- *Influence*: The dynamic should be able to be mitigated or resolved by the project or program office staff, rather than requiring action by Congress or a military service chief. While the problems requiring high-level intervention are no less interesting, the practical difficulty with resolving such issues makes them of less direct relevance to the acquisition community.
- *Scope*: The dynamic occurs within the context of a single acquisition program (rather than requiring the interaction of multiple programs)

The "Evolution of a Science Project" dynamic displays all of the above characteristics.

3.2 Acquisition Program Example Narrative

The following is a narrative of the history of an actual "science project" program. The quotes provided came from direct interviews with various program participants and stakeholders that took place over the course of an Independent Technical Assessment (ITA) conducted by the SEI. The specific names of individuals and organizations, as well as the nature of the system, have been omitted to protect the privacy of those involved.

Getting Off the Ground

The project began with a critical operational need for a new command and control system. One of the combatant commands was the service sponsor, and in the late 1990s they provided $250,000 in funding to the agency to build it—which got it started as an Advanced Concept Technology Demonstration (ACTD) project. The operational process was being accomplished with a telephone, paper, and pencil at the time, and as members of the operational community noted about the new system, "This fills a huge black hole," and it is "…important and needs to get to the warfighter soonest." The value of the system was clear, and there was no dispute that this capability could be a life-critical system that people would be counting on.

In its original incarnation "The software started out as a prototype, launched by [the agency]," and given the small amount of initial funding, "…it was originally a proof of concept…" designed to demonstrate the potential value of fully automating the process.

The project "…was not originally developed by any kind of acquisition center. It was funded by the user community" to solve a specific and urgent problem. This was done in part to avoid a lengthy formal acquisition effort that might take years, and instead field a small, but useful system more quickly. The agency took the money, found a "good prototype shop" to do the work, and "…probably someone had an existing contract vehicle that was used—it was a contract of con-

venience." It turned out that "The development contract was a T&M contract," for time and materials—a service contract type used when the work is expected to continue at some level for some period, and that generally isn't intended for producing a deliverable system.

The contractor began work building the prototype system for the agency. From the beginning the operational command had an informal management process, and in working with the command the contractor was "…happy with an 'agile'[3] approach, just discussing the requirements, with minimal documentation." In fact, the prototype system "…was never wholly designed against a complete set of requirements. As a result, some of the normal standard procedures for a program weren't applied." As work proceeded, it started to become apparent that "…the software had no design, no architecture," and "It was never documented what their architecture was."

The size of the prototype grew steadily as development continued, but without clear requirements or a documented software architecture to help them achieve their goal. As an early project manager observed, "They never did a true design for [the system]—they just began coding."

Pushing the Envelope

When the pressure is on to get a system deployed due to an urgent need from the field, a common consequence is that quality processes may be undermined in the rush to get the capability out. As the project manager ruefully acknowledged, "It's always possible to convince operational people to take shortcuts when lives are on the line."

Nevertheless, the contractor was able to produce a stripped-down prototype in 2001. A second version of the prototype system was first deployed in theater in 2004, and was well received. The lieutenant colonel at the combatant command who paid for it "…thought it was great, and had it installed in the command, and they loved it, but it crashed a year later, and then they hated it, and couldn't get it to run again." Still, proponents said "We had software that the user community likes, and saw work, and they want to know where we are with it." The project acknowledged that "We're late to need."[4]

As more time passed without a viable system in the field, members of the operational community were starting to threaten that they would have to file a Joint Urgent Operational Needs Statement (JUONS) to communicate the urgency of the need, which "…is like saying, 'Please, I *have* to have this now, or everything will fall apart.'"

In the contractor's race to add the needed capability to the system, the software complexity continued to grow, and code quality declined, making new development even more difficult. According to the government project manager the contractor "…admitted that their coding was a hodgepodge. They said that because they'd worked on the system for so long, they just kept adding

[3] It is more likely that an informal development approach was used that did not actually follow the tenets of agile development. The lack of documented requirements does not accurately describe the agile development approach.

[4] The phrase "late to need" means that the project is behind schedule in delivering a system that can meet the needs of the users.

things." The lead software developer confirmed that "Doing the coding was extremely rushed, there was no time for analysis, and no architecture before they did it (or at least only a very quick architecture), because of that original deadline."

At the same time that the number of defects in the system was rising and the quality of the software was starting to be called into question, the government project manager had designated the system as Mission Assurance Category (MAC) I—the highest level of criticality. As a senior software developer observed, "With MAC I, 'If the system goes down, someone dies'—so now you need an extremely robust system."

Stall Warning

As development continued, even the contractor software development lead felt that the code "…was an average technical product," and that due to the schedule pressure and circumstances it "…isn't the best thing you've ever seen." The poor code quality was further complicated by a lack of system documentation that made it even harder to work with, since in the rush to produce the code there was "...no documentation on the 'as-is' implementation in the software," making maintenance of the prototype system that much more difficult. According to the program manager, "When you have 700 KSLOC of undocumented code, what's your risk?"

After v2.5 of the system was deployed in 2008, the strain of supporting the fielded version of the system, while at the same time developing the features for the next version to be released, started to unravel the effort. The operational community representative summed it up this way: "Every time a problem came up, to make a change to [the fielded system], there was a conflict in priority with [the fielded system] vs. [the new system], so whatever money the project office gave [the contractor] for v3 development, it didn't go to actually doing new development—it went to [the fielded system] to fix older problems." The project manager concluded that "The issue with [the contractor] was that they couldn't do both sustainment and development well at the same time—they could only do one or the other." Deep into the development, it became clear that with respect to the contractor, "They weren't delivering."

As the new v3 development work stalled while the contractor team was spending their time fixing the problems with the deployed system, things began to come to a head. The project manager at the time concluded that "The concept was great, and everyone loved it, but the program fell behind, went over budget, nothing was getting produced, there were contractor non-performance issues, and it became a black hole for money."

Hitting the Wall

The operational community knew that the program had to be scaled up and formalized to be able to deploy a robust system to the field. "The [agency] commander was getting hammered by the generals in theater for [the system] not being properly operated and maintained, and said 'We're not an O&M facility—it should go to someone else now.'" The operational community had already realized that things were off track with the structure of the program, saying "ACTDs are supposed to move out and become real programs after a year or two—but [the agency] kept [the program] much longer than that."

The program had to be "shopped around" to several different organizations until one was found with the acquisition experience to be able to initiate a full PMO, and treat it as a formal acquisition program. Given the checkered history of the system up until then, there were few organizations willing to take on such a troubled program, with a contractor that had little acquisition rigor or experience. Even though they were finally convinced that the program had to be handed off to an acquisition organization for its own good, there were still misgivings in the user community—they knew that "the [formal acquisition] processes are very thorough, but they're slow, so the operational users hated it, and the time it took."

When the PMO stood up, the new Program Manager (PM) knew that he had his work cut out for him. He realized that continuing to have the contractor build the system on top of the prototype's foundation may not be a viable way to move forward. In assessing the contractor's ability, he said "They're not easy to work with… I'd characterize them as a prototype shop… they haven't demonstrated the necessary discipline to develop and field a system."

This lack of software development discipline on the part of the contractor became a recurring theme as the PMO's relationship with it evolved. In making the move to an official acquisition program, they "went from [the operational command] where they had a 'loosey-goosey' process with little documentation, to [the PMO] where they had formal system/software engineering." The change was dramatic for the contractor, given that "…they never transitioned from [being an] ACTD to a formal… program of record." In the eyes of the government test people supporting the system, the PMO was insistent that it be a formal Acquisition Category (ACAT) program, and as a result was "…trying to make [the contractor] play by two separate sets of rules—ACAT vs. ACTD... [There's] a mismatch of the requirements between those two different types of programs."

The contractor struggled to try to meet the demands of the PMO and the more formal ACAT rules it was now subject to—but it did not go smoothly. Even as new defects were being inadvertently injected into the code as rapid incremental development proceeded, the PMO testing lead observed that "The contractor hasn't been good about [tracking defects]… at least with any discipline or rigor." In general, the PMO said that "Trying to push them to use formal development processes was hard—and upper [contractor] management resisted. They didn't see why it was necessary… [They] said they weren't required to do that." As one contractor PM summarized it: "Up till then, everything was operationally focused, on getting it out to the warfighter—do whatever you have to do. But [the PMO], being more of a materiel command, said, 'You did good work on the product, but a terrible job on the process to make this a formal program. You did a horrible job on acquisition.' So they [the PMO] said, 'I don't care if the system dies, I just want the documentation'—which was 180 degrees different from what they did before." While the contractor thought the PMO was being unreasonable, there was reason for concern, as the contractor's "…track record with documentation had been in the gutter," and yet the PMO was legitimately "…worried about maintainability in the future with the lack of documentation." This was no small matter to them, because the PMO was already seeing issues with the operations and maintenance of the system, acknowledging that "The O&M side of this is a killer. We're spending millions and millions on help desk and operations…"

With this in mind, the new government PM began to review his options: move forward with the existing code base and retool and document it on the fly, or start over with a new architecture, and rebuild the system essentially from scratch. When the operational community got wind of this they became very concerned, saying "…The PMO wants to [can] what they've got, and start over. [The operational user community] doesn't want to just throw away what they have—and it'll be years until we can get something this good." Given that they finally had a system that was close to doing much of what they wanted, they felt that too much time had passed to scrap the system. They wanted "…an 80% solution people can use," and told the PMO directly that "The user community is… unwilling to wait 3-4 more years for a new replacement system to be built from scratch."

The operational user community had reason to be concerned about the PMO's decision, because even the new PMO didn't have all of the software and technical expertise needed to figure its way out of the hole that had been dug for it. Both the operational people and the contractor agreed that "Their [PMO] engineers don't have the software expertise that they should have, and don't have operational expertise," and that the PMO "…was *not* an extremely technical group…"

Crash and Burn

The conflict between the PMO and the contractor continued to escalate as progress on the new v3 development continued to slow, almost coming to a halt entirely. As far as the new development work went, the PMO was realizing that "…there was not enough there to field the system. They worked with them for a year and a half, and they weren't getting there."

The operational community representative observed in frustration that "I can't figure out how they've spent so much money on something this simple."

The PMO decided that the problem with the system was fundamental—it couldn't do its job "…because the system design to do that wasn't ever completed." The PMO believed that due to the lack of a coherent architectural vision, the software as built could never be successfully fielded in theater. They felt that for any problem that came up, the contractor's "…solution was to throw more hardware at it"—up to "$750,000 at each theater site," and that what they had developed was a "…really clumsy solution, and consumes inordinate amounts of bandwidth." Given a choice between requiring infeasible amounts of hardware at each site, or consuming excessive bandwidth between each theater and the systems' central servers—the PMO decided to choose neither.

The PMO finally issued a "stop work" order for the new development work, and allowed the task order to lapse—effectively terminating the development contract with the contractor, more than 10 years after development had first begun.

Despite the operational community's objections, the PMO decided to discard all of the software that had been developed over the preceding decade, and begin a complete redesign of the software. Its plan was to develop an entirely new architecture to create a next-generation system with even more capabilities than the one they were discarding. This was an effort that would likely cost tens of millions of dollars more, and, as the operational community predicted, take an *additional* 4-6 years—if everything went perfectly…

3.3 Qualitative Analysis of the Dynamic

Software-reliant acquisition programs are inherently complex, and in this respect "The Evolution of a Science Project" is no different. The following sections discuss some of the key aspects of the dynamic in greater detail, describing how each one is believed to contribute to the occurrence of the dynamic in actual programs. These qualitative aspects of the dynamic were elicited through repeated detailed discussions with acquisition domain experts, and were later incorporated into the system dynamics model.

3.3.1 Misaligned Incentives and Structural Dynamics

The dynamics of "The Evolution of a Science Project" are based on two types of underlying forces: misaligned incentives and structural dynamics.

Broadly speaking, misaligned incentives are aspects of the acquisition system such as its rules, regulations, policies, and guidelines that come into conflict when an acquisition decision-maker must trade-off maximizing the achievement of one objective against maximizing the achievement of another. For example, misaligned incentives in acquisition programs can put individual or program-specific interests ahead of PEO or Service interests, turning planned cooperation into opposition, and producing poor acquisition outcomes.

Structural dynamics, however, are the natural, or even in some sense physical aspects of developing and acquiring software-reliant systems. These structural aspects include such components as feedback loops and time delays.

As an example of the combination of both misaligned incentives and structural dynamics, we can think of the situation in which strong schedule pressure is exerted upon a software development manager by higher-level management and other stakeholders. The amount of that pressure that the manager then chooses to place upon the developers is a management decision that he or she makes based largely upon the incentives at work: the potential effects of late system delivery on his or her career, the expected effects of the schedule pressure on the developer productivity and morale, and so on. If the manager chooses to exert substantial schedule pressure on the developers, then the ongoing and/or rising schedule pressure will at first increase, but ultimately degrade productivity as developers begin to "burn out" and leave the organization—a structural dynamic that contains both feedback and a time delay. While the amount of pressure that is placed on the developers is a management choice that is the result of incentives, the results of chronic schedule pressure on the developers are not a choice—they are predictable and almost inevitable.

Taken together, misaligned incentives will drive decisions that seem (and may be) reasonable at an individual level, but they may also have unintended consequences. The intent of increasing schedule pressure on developers may be to *raise* their productivity to meet a schedule—but the longer-term consequence is likely to be the opposite. The manager may make a calculated gamble that the consequences to himself or herself personally of trying to accelerate development through increased pressure on the developers will only become problematic *after* the next key deliverable is made, after which schedule pressure will decrease, and thus the productivity benefits will be gained without suffering the potential costs of developer burn out and turnover.

With both types of forces present in many different acquisition dynamics, their interdependence injects additional complexity into the behavior. "The Evolution of a Science Project" dynamic displays both aspects in abundance, and the two interact to produce the counter-intuitive and counter-productive outcome that we often see.

3.3.2 Incentives Driving "The Evolution of a Science Project"

There is a confluence of misaligned incentives that is at the heart of "The Evolution of a Science Project" which sets the dynamic in motion toward failure, and keeps it moving in that same direction. There are five incentives that are most significant, and these are listed below, along with simulated "quotes" that suggest the likely thinking of those affected by these incentives.

- *Incentive*: There is a very real need to get a capability into the field as soon as possible.
  - "We have to do *something* about this problem for the warfighter, and *soon*…"
- *Incentive*: It's difficult to make a compelling business case to get funding so that the new system concept can be tried out—without a prototype/demonstration
  - "…but we'll never be able to get funding to do it without having something to show people…"
- *Incentive*: Formal acquisition is known to be slow, and has a high failure rate—so every effort is made to avoid that approach.
  - "…and we *can't* rely on formal acquisition, which is too slow, and will likely fail,…"
- *Incentive*: The operational/business people believe that they should manage it, since they know the domain better than anyone—and much better than the acquisition people do
  - "…so we can do it *better* ourselves, since we already know the domain,…"
- *Incentive*: The operational/business people in charge of the project believe they can build the system more quickly than the formal acquisition process because it's "only software."
  - "…and software isn't that hard to develop—we can put this together ourselves (or with a little help) in six months!"

What happens is that these incentives inevitably create the following near-ideal initial conditions for initiating "The Evolution of a Science Project" dynamic:

- led by business/functional people
- lack of large-scale software development experience
- prototype development
- schedule pressure
- resistance to formal acquisition

3.3.3 User Demand, Satisfaction, and Schedule Pressure

User demand is at the heart of "The Evolution of a Science Project" because the needs of the users bring new projects and programs into existence. User demand is present long before the concept of schedule pressure comes into existence. As development continues and new features are added, this affects both user demand and user satisfaction, which influence schedule pressure. Actual

software development programs make no attempt to measure schedule pressure, thinking of it only in terms of the relationship with schedule delays—as the schedule slips, schedule pressure generally increases. Even though schedule pressure is widely acknowledged to be a key driver in program behavior and performance, little attempt has been made to understand it. This is problematic when creating a system dynamics model, as schedule pressure is also affected by user demand, as well as by user satisfaction. Developing (and deploying) new features should satisfy users to some extent, reducing demand. However, the delivery of new features also makes users realize how much more capability they *could* have, thus *increasing* user demand.[5]

3.3.4 User Community Management

Science projects are often initiated—and even frequently managed—by their user community, who are likely experts in their relevant domain area, but generally lack expertise in software engineering and management. Some projects may be organized as less formal "in-house" developments, while others are begun using contractors who specialize in building advanced prototype systems. Still others are the product of laboratories or Advanced Concept Technology Demonstration (ACTD) efforts. Regardless of the formal structure, science projects often evolve organically without the structure or investment in design that should be put into building a large, mission-critical software-reliant system. These short-cuts sometimes occur because there isn't yet adequate demand for the capability to justify a formal program. In other instances there has been a conscious decision to try to develop it as quickly as possible by "flying under the radar" and avoiding the bureaucratic overhead and cost of the formal acquisition process. In these cases the operational community may not have the acquisition experience to understand and appreciate the longer-term value of such practices as good process adherence and quality system documentation.

3.3.5 Uncontrolled Prototype Growth

A potential downside with science projects is that these systems can become the victims of their own success. The natural and laudable instinct of commanders to provide valuable new capabilities to their warfighters in the field as quickly as possible can become a threat to the project. The initial prototypes of science projects often grow incrementally without a guiding vision or architecture and are informally tested in the field, often with very positive initial reactions to the system's new capabilities. Once the value of the system's capability is recognized by warfighters and other users, demand increases quickly, and the warfighters may soon become unwilling to part with these new capabilities. The systems have already incorporated many new features in response to warfighter needs, with the code base becoming increasingly hard to maintain, and defects being injected as each change is made or new feature is added. Documentation—rarely a strong point of most prototype development efforts—is no longer an option.

In such an environment, science projects are often forced to evolve rapidly into full-scale acquisition programs that can reliably deliver a robust production system—a growth spurt that was rarely anticipated or properly planned. It is at this point that many science projects "hit the wall" with a large, mostly undocumented, convoluted, defect-laden, and unmaintainable code base, unable to make progress at their early development rates. Almost any change made to such a system now

[5] The way in which the system dynamics model reconciles these two conflicting forces is described in Section 4.2.1 in the discussions of the R1: Escalating User Demand and B3: Moderating User Satisfaction loops.

will have multiple adverse and unintended consequences on quality, performance, and robustness. The software, like the plumbing in an old house, can only be patched and repaired for so long before the entire system must be scrapped and re-implemented.

3.3.6 Project Manager Decisions

There are several parts of the model where the input is the result of a project manager's decision. These types of inputs can be modeled in two ways, depending on the purpose of the model: 1) creating a constant or function (based on other system variables) in the model that behaves as the manager would, or 2) allowing the user of the model to play the role of the manager, and provide the input as they see fit. The first approach is generally taken when the model is being used for analysis of the dynamic, and the second when the model is being used for interactive training of a student. Some examples of such points in "The Evolution of a Science Project" model include:

- the manager's decision as to how much of the schedule pressure should be transmitted down to the developers (a good manager will likely want to shield developers from more than a small amount of pressure to avoid burn-out and turnover—but while it is desirable to believe that managers don't push their developers to the point of burnout, there is significant evidence to the contrary)
- the manager's decision to reassign developers to do rework as more errors are found (a good manager will want to maintain system quality standards even at the expense of schedule delays, but this is an especially difficult stance to take in the context of a science project development)
- the manager's decision as to the amount of calendar time they are willing to spend on each rework task (a good manager will keep this amount constant, but there may be a temptation to reduce it as time is running out before a deadline)
- the manager's decision as to the allowable threshold for code quality or defect density (a good manager will keep this value fixed throughout the duration of the project—but as the manager's options dwindle in the face of growing delays, robustness issues, and increasing demand, further sacrificing quality becomes an increasingly attractive alternative)
- the manager's decision as to when the science project prototype development should end, and formal production development should begin—which affects not only how much capability has been developed at that point, but also the quality of the software, the defect density, and the amount of remaining effort

3.3.7 The 90% Syndrome

In their book "Software Project Dynamics" Abdel-Hamid and Madnick described an effect that was already widely known as "The 90% Syndrome" [Abdel-Hamid 1991]. This dynamic occurs when a project's planned effort is underestimated, and progress (in terms of "% complete") is difficult to quantify, and can't be measured directly except through the ongoing consumption of time and resources. As the project nears completion the difference between the perceived progress and actual progress becomes clearer, and a more accurate picture of the remaining work emerges. Part of this remaining work is the body of undiscovered rework that has been invisibly accumulating as defects are injected into the software, but are missed by testing. As the understanding of the

amount of work and rework left to be done encompasses more tasks, the measure of progress to date should actually *decrease* from the prior estimates. However, since progress is rarely (if ever) described as being negative, the measurement of forward progress only stalls, often in the 80-90 percent complete range, while the remaining work is seen to expand well beyond the expected 10-20 percent of the original effort. Measurement of progress then advances slowly (even if true productivity actually *increases*) through completion as actual development progress catches up with the prior progress estimates.

The early circumstances of "The Evolution of a Science Project" lend themselves well to creating a "90% Syndrome" dynamic, both because of the informality of the development and the accompanying lack of metric data collection, and because of the typically higher levels of defect injection due to the comparatively rapid pace of development.

3.3.8 Re-Architecting the System

As major new capabilities must be added to the prototype to satisfy the still-growing user demand, the project's managers can often see the shortcomings of its evolved design and may want to pause development to re-architect the system. Insistent user demand, however, won't allow the project to "go dark" for months—much less years—to do work that will produce little in the way of visible new features for the end users. Even if the government were still inclined to discard the prototype and start new development from scratch, the contractor has an incentive to persuade the government to reuse the prototype software that the contractor has built as the platform for the new work, thereby making the contractor more indispensable to the future development effort (since starting development over would create a more level playing field on which other contractors could compete for the work).

3.3.9 Project Management Infrastructure

Often, when the technical aspects of the development effort encounter difficulties, the management aspects do as well. The project infrastructure (the project team size, its processes, the level of software development and management experience, etc.) that may have been adequate for building a prototype can be inadequate for developing a formal production system intended for wide deployment. These mismatches can mean that most aspects of the project are now inappropriate for their new purposes and must either be discarded, or substantially changed and expanded.

Science projects frequently end up morphing awkwardly and ineffectively into convoluted production systems due to the desire to deploy new capabilities to the field. While not inevitable, in an environment of scarce funding and chronic schedule pressure, there is a strong temptation to continue development on the same code foundation, with the original project infrastructure, in the often mistaken hope that as more and more defects are fixed, the software will eventually become stable and robust enough to suffice for use as a production system.

3.3.10 The Transition from Science Project to Formal Development Program

While the precise location of the line between the existence of a science project—as opposed to its transition to a formal development program— may not always be clear, they are inherently

different phases with different behaviors. Thus, there is good reason to explicitly separate out the modeling of these different project phases. While many aspects of the dynamics of conducting software development are similar (they are both performing essentially the same activity), the science project and production development phases still have distinctly different characteristics. Ideally, the two different phases of the project should be modeled using a single model with different parameters to represent the different aspects of the two development phases (so as to ensure consistency), but this was not possible in the timeframe available for this first increment of the work. As a result, the models used for the two phases are different in ways that reflect the different management and development philosophies used. The conceptual connection between the two models is represented by having the formal development program inherit the undiscovered rework from the science project, as well as the completed software development tasks and the remaining schedule.

3.3.11 Evolutionary Prototyping vs. Throwaway Prototyping

An important aspect of the "Evolution of a Science Project" dynamic is the distinction between *throwaway* prototyping and *evolutionary* prototyping [Crinnion 1991]. As the names indicate, throwaway prototyping describes the creation of a system with the intention that it will be discarded at some point in favor of the development of a production-quality version. Throwaway prototypes are used to demonstrate proof of concept, and to show prospective users how the system is expected to appear, and perhaps to work, in large part to clarify requirements. Refining requirements through a throwaway prototype is cost-effective, as the cost of repairing requirements errors later in the development process is expensive. The development approach for a throwaway prototype is typically informal, with little in the way of a formal architecture, coding standards, or system documentation, and with speed of development being a key goal.

In evolutionary prototyping there is a different goal—to build the prototype in a more formal and structured way with the intent of "evolving" it incrementally into a quality production system. This evolution is the basis of a number of development approaches, including agile methods, but all such methods still require software expertise and rigor to ensure the quality of the evolving prototype. While the evolutionary prototype shares with the throwaway prototype the idea of continuing requirements elicitation and refinement, as well as incremental development, the approach to doing so must be very different. The architecture of the evolutionary prototype must be sufficient to support and maintain a full production system, but the throwaway prototype should bear no such burden, as it will be discarded when the requirements are adequately understood. The evolutionary prototype, being a working system, can be deployed for use by users, even in partially complete form. This deployment often results in a stream of defect reports and further enhancement requests as the users become increasingly familiar with the system through regular use.

The differences between these two types of prototypes are important because they are a significant part of the basis for the "Evolution of a Science Project" dynamic. Since the science project is generally initiated by operational staff with minimal knowledge of either software engineering or formal acquisition methods (potentially working with a contractor with similar limitations), they are unlikely to be aware of the fact that not all prototypes are developed in the same manner. This means that due to schedule pressure the staff are likely to begin development as though they are building a throwaway prototype, but with the idea that if it is successful, they will be able to continue to develop and then deploy it as a fully functional production system—as though it were an evolutionary prototype. The staff will try to transform the prototype into a full-scale system that will be built on top of an unsound foundation. Without sufficient software engineering training or experience, this may seem like a viable approach, but will set them on a path toward growing problems with reliability, capability, performance, and usability—a path that leads inexorably to slowing progress and decreasing robustness.

4 System Dynamics Model and Simulation Results

This section describes the model for "The Evolution of a Science Project" and some simulation results.[6] Figure 1 depicts the basic system dynamics (SD) model development process. The process is iterative such that later steps in the process may iterate back to previous steps.

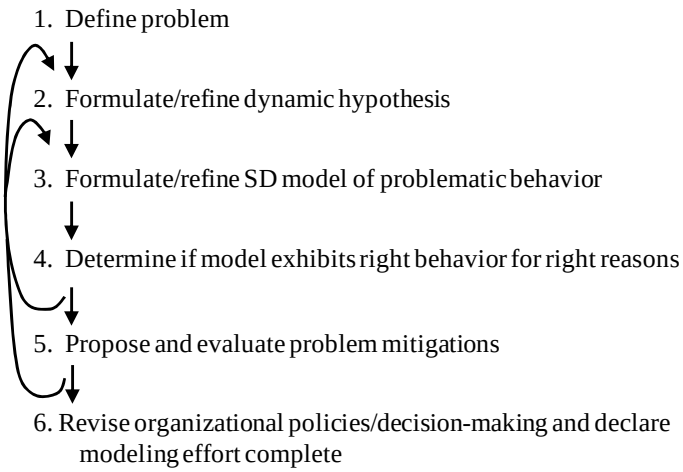


Figure 1: Overview of Model Development Process

This report describes the preliminary progress in developing the model through several iterations, with the current state of the model reflecting progress in step 4. The first step – "Define problem" – was accomplished in the first three sections of this report. The rest of this section describes the results of the simulation modeling effort as it progressed through multiple iterations of steps 2 through 4. Section 4.1 describes the causal hypothesis for problematic behavior in science projects. This hypothesis forms the basis for the simulation model described in Section 4.2. Finally, Section 4.3 discusses preliminary simulation results which provide tentative support for the causal hypothesis.

4.1 Dynamic Hypothesis

We hypothesize that the root cause for the problematic behavior experienced during the transition of many science projects is due to the fact that

- software development of the prototype is driven by schedule and the development of new features, with little focus on quality and robustness.
- the code from the science project prototype is used as the foundation for the production system because of the perception that starting over would unacceptably delay deploying new capabilities to the field (the architecture and design of the prototype might also be used, but in many cases these do not exist, and when they do, they are frequently poorly documented).

[6] For those who are unfamiliar with system dynamics, Appendix B provides a brief introduction to the subject that may provide useful background for reading this section.

- the transition from prototype development to production system development is delayed later than necessary, both to try to field a system more quickly, and perhaps consciously, to avoid the effort and overhead of formal program rigor.

Early in the life cycle of a science project, development results in a working prototype that has attractive external capabilities, but also high defect rates, poor code quality (i.e., complex and poorly structured code), and poor system documentation. The use of the prototype as the basis for a production system and the late transition of the prototype to production development inject a significant level of latent defects into the system. When discovered incrementally later in development, the latent defects and poor code quality lead to a "ripple effect" of rework across the phases of the development [Cooper 1980]. This creates schedule pressures in the production development that can magnify the ripple effect further. The ripple effect is depicted in Figure 2 below, notionally graphing discovered quality issues during production development over time. Discovered quality issues rapidly increase to a peak, in many cases leading over time to additional consequential issues—and then gradually decline as development resources are reallocated to rework them, with this rework itself introducing a smaller level of quality issues that will have to be reworked.

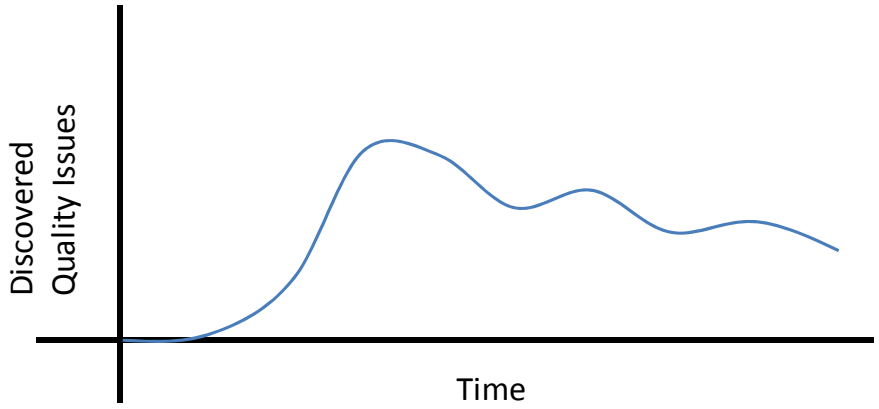


Figure 2: Notional Problem: Ripple Effect Due to Incremental Discovery of Science Project Quality Issues

We believe that the ripple effect described above can cause the "90% syndrome," as depicted in Figure 3, where latent defects that have accumulated during the science project phase slowly come to light during the production development in the later stages of the project. During the latter stages of development a much larger percentage of the effort is devoted to rework rather than development. That situation, combined with the fact that undiscovered work comes to light only incrementally over the lifetime of the project, results in the slowed progress inherent in the 90% syndrome.

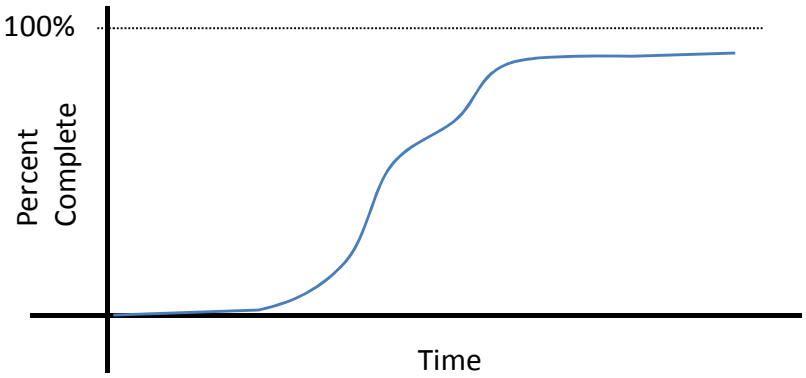


Figure 3: 90% Syndrome Due to Rippling Rework in the Production Development

Figure 4 (depicted on the next page) presents a causal loop diagram[7] that refines the causal hypothesis described above. The left side depicts the primary aspects of the development of the science project prototype. The right side shows the transfer of that prototype to production development. The development of the science project is motivated by urgent demand for the new capabilities in the field. The purpose is to build a small prototype system to produce a compelling "proof of concept" demonstration to help garner support for full-scale production development.[8] The project is often (although not always) initiated and managed by the user or operational community. The software development is often done in-house by existing staff, and if a contractor *is* used the contract vehicle may be Time and Materials (T&M) or Indefinite Delivery/Indefinite Quantity (IDIQ) without explicit schedules or deliverables, in order to keep the development effort as informal as possible. The effort is generally both schedule and feature driven to get key functionality to users as quickly as possible. The (blue) feedback loop on the left side of Figure 4 shows that schedule pressure can result in a reduction in the rigor of QA processes, with a commensurate reduction in the staffing of rework to fix quality issues in the prototype. This results in more effort being put into developing features and a reduction of schedule pressure compared to what it would have been otherwise.

[7] A causal loop diagram shows qualitatively how related variables affect each other. The nodes indicate variables, and the connecting arrows show the nature of the relationships between them. When an increase in one variable causes an increase in the variable it is connected to, the connecting arrow is marked with a "+". When a change in one variable causes the *opposite* change in the variable it is connected to, the connecting arrow is marked with a "-". A loop made up of nodes and connecting arrows that continually moves in the same direction is called a "reinforcing" loop. A loop where the values of the nodes oscillate back and forth is called a "balancing" loop. Two parallel lines placed across a connecting arrow indicate that there is a delay in the propagation of the value across the arrow [Kim 1993, Kim 1994]. A more detailed discussion of the notation for causal loop diagrams may be found in Appendix B of this report.

[8] It should be noted again that this scenario is *not* what would be done when following an evolutionary prototyping approach in support of agile development method, where formal steps would be taken to ensure the quality and robustness of the evolving prototype.

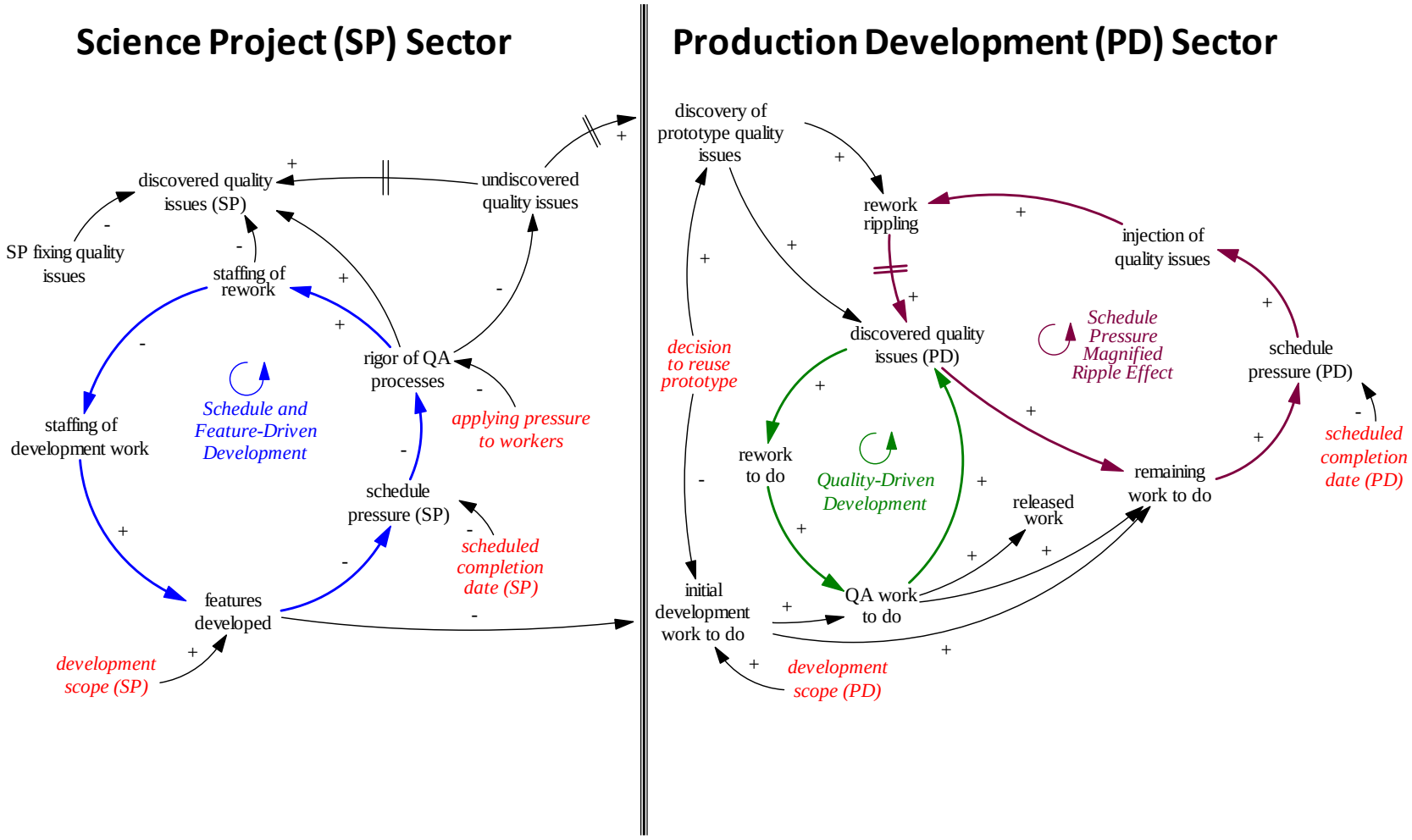


Figure 4: Causal Loop Diagram of "The Evolution of a Science Project"

While reducing the rigor of the QA processes enables more rapid development of features, it also has clear repercussions on the quality of the prototype as depicted in the upper portion of the Science Project sector. Reduced QA rigor leads to the generation of more software quality issues, which are initially unknown (or undiscovered) by the project team. It also makes it difficult to discover and fix these quality issues (fewer QA inspections, etc.) when staffing of rework is diminished. The result is that quality issues are rampant in the prototype when the decision is made to transition the system to a formal acquisition program, especially when the project resists making that transition as soon as possible after proving viability.

The decision to reuse the prototype in production development may be irresistible. As was pointed out previously, having operational people manage the project instead of software development or acquisition professionals may prevent them from seeing the longer-term consequences of using a prototype as the basis for production development. Upon the decision to reuse the prototype, as shown on the right side of the Production Development sector, the production development inherits the work that has been completed on the prototype. Unfortunately, it also inherits the prototype's quality issues. Most significant is the large stock of undiscovered quality issues. While the production development adopts a quality-driven development approach appropriate for a formal acquisition program, as shown in the (green) feedback loop, the discovery of prototype quality issues creates a ripple effect of rework across the phases of the production development.

Of course, the production development effort is not immune to its own schedule pressure. While some level of defects was expected, the incremental discovery of latent prototype quality issues stemming from significant architectural and design flaws was not fully considered in the schedule for the production development. The large amount of rework caused by the ripple effect of the incrementally discovered quality issues throws the production development off schedule, thus increasing pressure on development staff and as a consequence, increasing the injection of quality issues during development. This effect magnifies the ripple effect further as seen in the (purple) feedback loop on the right side of Figure 4.[9]

It might appear that if the program management simply did a better job of planning for the additional rework caused by the prototype, the problems could be resolved. However, it is the existence of fundamental issues with the prototype software's architecture and design that led to much of the undiscovered (and unsuspected) rework that will have to be done—and the effort that will be involved to address those issues will be both substantial, and especially difficult to estimate. The conclusion is that the causal mechanisms described above may be the determining factor behind the poor performance of many science projects even after they have transitioned to a formal acquisition program.

4.2 Simulation Model

A quantitative system dynamics model refines and describes the relationships presented in the causal loop diagram using mathematical equations. This is done by adding two new concepts to the modeling notation: stocks and flows.

[9] Note that in the diagram QA Work to Do has the same influence on both released work, and remaining work to do. This is simply a reflection of the fact that as the amount of QA work increases, it results in both more released work, and more remaining work to do (i.e., rework).

1. Stocks represent accumulations of physical or non-physical quantities, and flows represent the movement of these quantities between stocks. Stocks are depicted as named boxes within the model.
2. Flows are depicted as double-lined arrows between the stocks with a named valve symbol indicating the name of the flow. Flows that come from (or go to) a cloud symbol indicate that the stock from which the flow originates (or to which the flow goes) is outside the scope of the model, and presumed to be infinite.

Figure 5 provides an overview of the system dynamics model. The three feedback loops used to describe the causal hypothesis in the last section also appear in this overview. The loops correspond with loops B2, R3, and R4 in the figure and are labeled with the same name and appear in the same color. The left side of the figure portrays the Science Project sector, and the right side shows the Production Development sector. The overview in Figure 5 provides an index to the structure of the full model presented in Appendix C and Appendix D.

The overview presents both the primary and secondary feedback loops that are most important for describing the evolution of a science project. At this level there are nine feedback loops—five balancing loops and four reinforcing loops.[10] The model embodies a simplified story of the evolution of a science project, which integrates the exposition of the feedback loops described in more detail in the next section:

> *Developers build features while reworkers fix quality issues. Developers may get pulled off development to do quality driven rework (feedback loop **B1**).[11] However, science projects are largely schedule and feature-driven to satisfy users, so managers resist staffing rework at sufficient levels (**B2**). While new features may ultimately satisfy users (**B3**), in the short term they create heavy user demand, increasing schedule pressure (**R1**). Increased schedule pressure may increase short-term productivity (**B4**), but at the expense of more quality problems. Even worse, workers eventually start to burn out due to the increased work load (**R2**). The steady erosion of the QA processes and increasing problems in the field creates pressure to move to a formal acquisition program.*
>
> *As the science project transitions to production development, that development inherits undiscovered quality issues in the science project prototype.[12] This creates a "ripple effect" of rework that is discovered incrementally through the production development (**R3**). All the while, the production development strives to approve and release work to increase the capability of the fielded system (**B5**). Unfortunately, schedule pressure during the production development mounts due to the incremental discovery of pre-existing quality issues (and thus*

[10] The terms "balancing loops" and "reinforcing loops" are described in Appendix B as part of an introduction to system dynamics.

[11] Note in the diagram that more SP Reworkers results in an increase in both reworking success *and* reworking failure. This is a reflection of the fact that as the number of SP Reworkers increases, discovered quality issues will be dealt with more quickly through rework done in either a correct (successful) or incorrect (failing) manner. While seemingly counter-intuitive, this is simply a consequence of the increased amount of work being done. We assume that there is no change in the percentage of rework success due to an increase in SP Reworkers.

[12] While it looks like SP Discovered Quality Issues have to be rediscovered in the production development, the quality issues that are discovered during the science project development are immediately available as discovered quality issues in the production development, while the undiscovered quality issues take some time to discover during production development. This is signified by the delay mark on the arrow from SP Undiscovered Quality Issues to PD Discovered Quality Issues. This behavior is apparent in the mathematical formulas representing the simulation model, but is somewhat ambiguous at this qualitative level.

*additional rework) from the science project. The ongoing pressure creates quality problems in the production development as well, and magnifies the ripple effect of undiscovered quality issues (**R4**).*

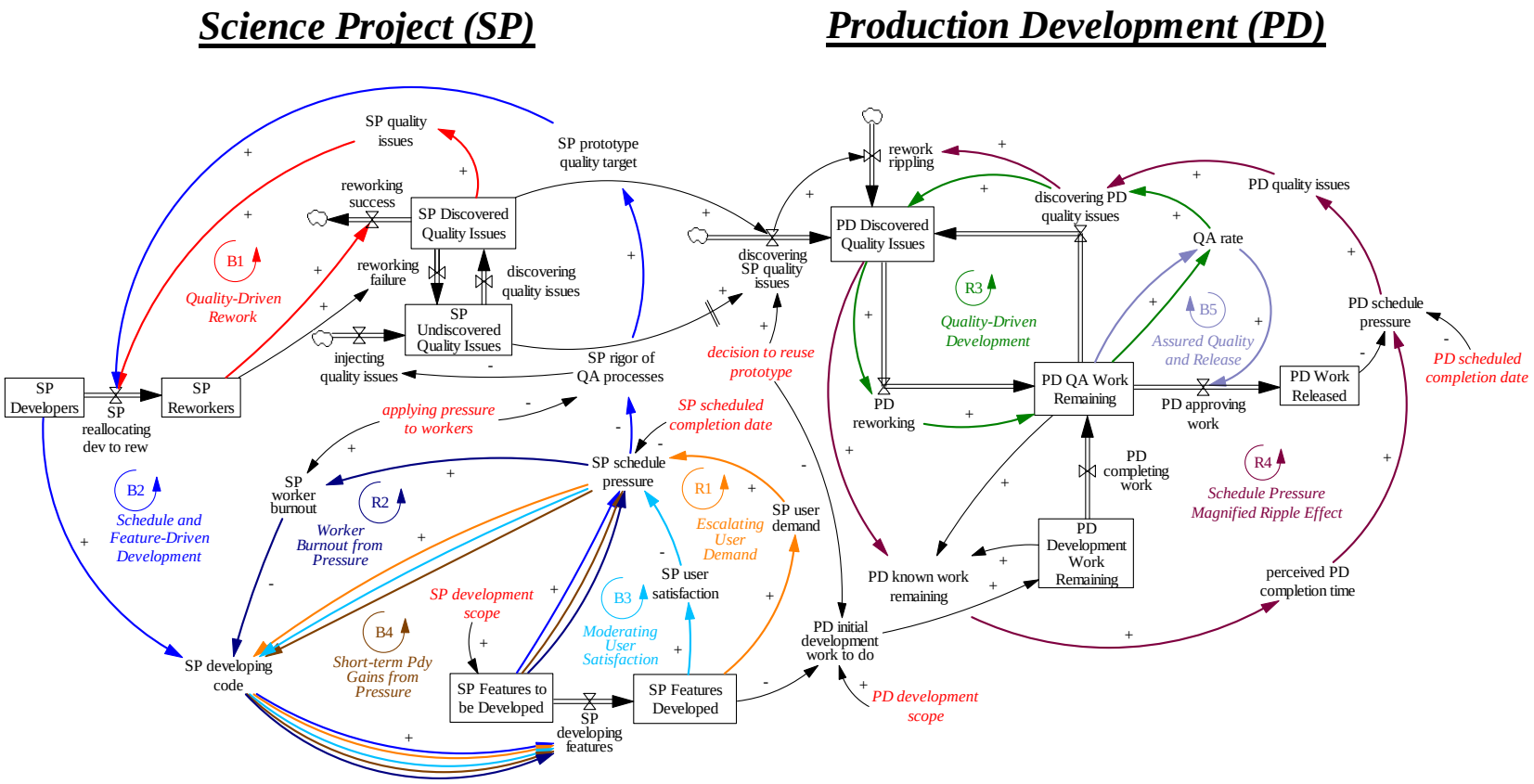


Figure 5: Overview of the System Dynamics Model

4.2.1 Science Project Sector

The key stocks and flows in the Science Project (SP) sector are depicted in Figure 6 and are understood as follows:[13]

- *Worker reallocation (along the left middle of Figure 6)*: Science project personnel are either developers, i.e., writers of code to implement features; or reworkers, i.e., fixers of quality issues related to code defects, code quality, or software architecture. We assume a fixed number of employees working on the science project prototype, but with full flexibility to move staff between development work and rework based on the priorities of the project, with a small time lag to actually make the transition. Note that while the flow is only shown by the arrow to occur in one direction, a negative flow results in reworkers becoming developers.
- *Feature development (along the bottom right of the figure):* The model starts out with a basic number of features to be developed. The stock and flow shows that features are developed at a rate represented by the "SP developing features" variable. In the model this rate is based on the number of SP Developers available and their (average) productivity,
- *Quality issue processing (along the top right of the figure):* Quality issues injected into the software during development are initially unknown by science project personnel. Over time these quality issues may be discovered and, depending on the number of SP reworkers available, the associated software will be reworked. The rework may be successful, resulting in the correction of the problem, or may fail, leading to additional undiscovered quality issues.

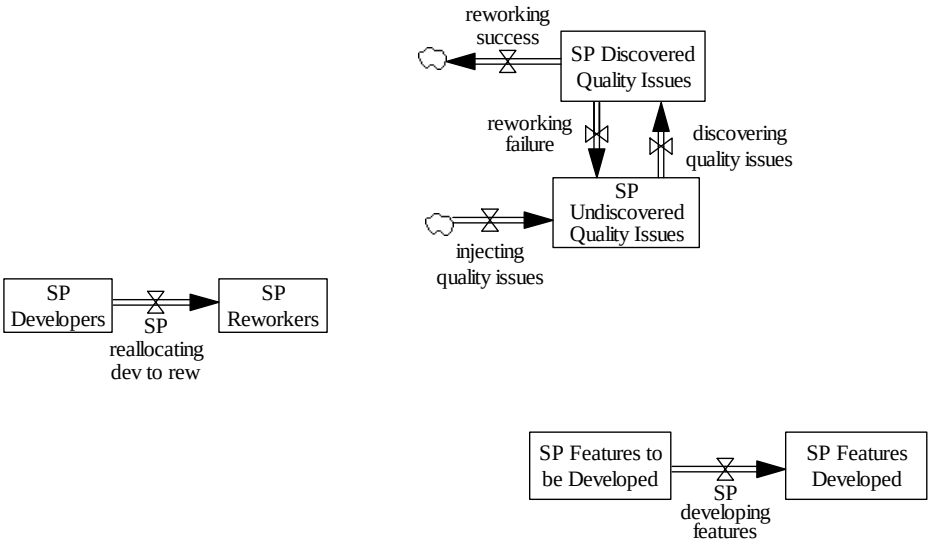


Figure 6: Stocks and Flows in the Science Project Sector

[13] While there are additional stocks and flows in the full model shown in the appendices, we believe that the ones describe here are key to understanding the basic dynamics of the model. By understanding this simpler model, the interested reader will be well on their way to understanding the full model.

The stocks and flows of Figure 6 are interconnected via the feedback loops shown in the Science Project Sector of Figure 5. The key feedback loops are:

- *B1: Quality-Driven Rework (red)*: This balancing loop represents the relationship in the Science Project between the number of people doing rework, and the number of discovered quality issues in the science project. It shows the feedback in attempting to reduce quality issues through reallocating people to do rework.
- *B2: Schedule and Feature-Driven Development (medium blue)*: This balancing loop represents the relationship in the Science Project of the number of developers, the code and features they develop, schedule pressure, QA rigor, and the prototype quality target. It represents balancing feedback in attempting to meet schedule by lowering QA standards, reallocating people to do development rather than rework (i.e., sacrificing quality to make better progress), produce more features, and reduce schedule pressure.
- *R1: Escalating User Demand (orange)*: This reinforcing loop represents the relationship in the Science Project between the number of features that have been developed, the user demand for more features, schedule pressure from users, and the amount of developed code. It shows the feedback effect of how more features having been developed actually increases the user demand for even more features, driving up schedule pressure to develop more code.
- *B3: Moderating User Satisfaction (aqua)*: This balancing loop represents the relationship in the Science Project that tends to reduce (somewhat) the schedule pressure, as users get access to features developed by the Science Project and to some extent are satisfied with those features. This loop, therefore, partially counteracts the R1 loop described above. Combined with the R1 loop, the cumulative result may still be that schedule pressure increases based on user reactions, but the B3 loop represents those influences that dampen the schedule pressure below what it would have been otherwise.
- *B4: Short-term Productivity Gains from Pressure (brown)*: This balancing loop represents the relationship in the Science Project between the amount of developed code, the number of new features developed, user satisfaction with those features, and schedule pressure from users for more features. It shows the feedback in attempting to boost productivity by pressuring developers to produce more code, thus relieving the schedule pressure at least in the short term.
- *R3: Worker Burnout from Pressure (navy blue)*: This reinforcing loop represents the relationship in the Science Project between the amount of developed code, the number of features developed, user satisfaction with those features, schedule pressure from users for more features, and worker burnout from the schedule pressure. It shows the feedback effect of ongoing schedule pressure producing burnout among developers, thus adversely impacting their productivity in developing code and increasing schedule pressure.

4.2.2 Production Development Sector

The key stocks and flows in the Production Development (PD) sector are depicted in Figure 7. The stocks are

- PD Development Work Remaining: This stock contains the development work to be done. Once this work is completed (at whatever level of quality the current processes afford) the work is sent on for quality assurance evaluation.
- PD QA Work Remaining: As development work is completed, quality assurance processes take over. This stock contains the backlog of QA work to be done.
- PD Work Released: Work that is approved is released into the PD Work Released stock. How the work is used once it is released is not modeled, nor is allowance made for the possible discovery of quality issues in released work products. For this model we assume that the quality assurance processes are sufficient to ensure acceptable use of the work products in the field.
- PD Discovered Quality Issues: Work that is found to have significant quality issues goes to the PD Discovered Quality Issues stock. Quality issues may also be discovered due to problems in the SP prototype, which has a separate flow into the stock of discovered quality issues. The rework rippling flow is yet another inflow to discovered quality issues that represents the potential for ripple effects of additional rework due to quality issues found in previously developed artifacts.

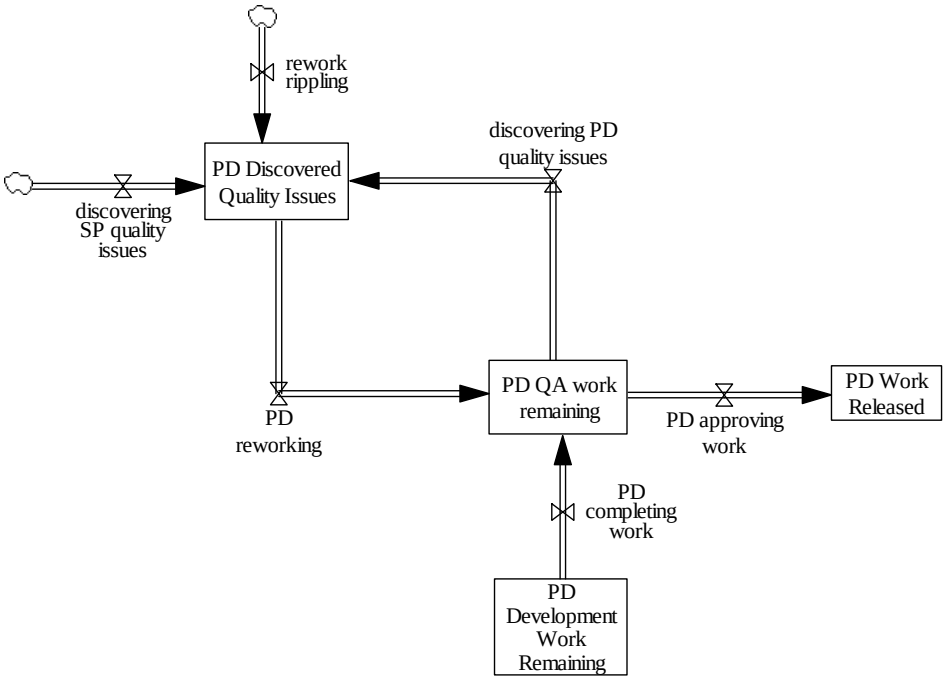


Figure 7: Stocks and Flows in the Production Development Sector

The stocks and flows of Figure 7 are interconnected via the feedback loops shown in the Science Project Sector of Figure 5. The key feedback loops are as follows:

- *B5: Assured Quality and Release (gray)*: This balancing loop represents the relationship in Production Development between the amount of QA work remaining, the rate at which QA

work is being accomplished, and the rate at which work is being approved. It shows the feedback of pending QA work that influences the rate of approving completed work.

- *R3: Quality-Driven Development (green)*: This reinforcing loop represents the relationship in Production Development between the amount of QA work remaining, the rate at which QA work is being accomplished, the discovery of quality issues, the rate at which rework is being done on those issues. It shows the feedback effect in which the discovery of quality issues as a result of the QA process causes rework, thus increasing the QA backlog in a reinforcing way. This is one dimension of the ripple effect of undiscovered quality issues.
- *R4: Schedule Pressure Magnified Ripple Effect (purple)*: This reinforcing loop represents the relationship in Production Development between the amount of known work that is remaining to be done, the perceived time needed to finish that work, the schedule pressure, quality issues, and the discovery of quality issues. It shows the feedback effect in which discovered quality issues increases schedule pressure, and magnifies the ripple effect described in the R4 loop.

Note that the rework (or "Firefighting") dynamic is evident in the Science Project sector in the *B2: Schedule and Feature-Driven Development* feedback loop, and in the Production Development sector in the *R4: Quality-Driven Development* and *R5: Schedule Pressure Magnified Ripple Effect* feedback loops.

4.3 Preliminary Simulation Results

The full model (which is described in the appendices, and was discussed in overview in the previous section) is implemented to provide a simulation of the entire science project evolution. Appendix E shows the interface that was created to interact with the model. This section presents a few preliminary simulation behaviors that reflect the team's experience with science projects in the field at a qualitative level.

A word of caution is necessary. The model has not yet been validated or calibrated to align with the experience of acquisition program decision-makers outside the modeling team. As such this simulation model is only a preliminary explanation of the dynamics of "The Evolution of a Science Project." The causal hypothesis described previously remains a hypothesis. The simulation model described here provides one avenue for testing mitigations to the problematic behavior, but work remains in terms of refining and validating the quantified relationships among model variables before testing mitigations would have significance in the real world. The model must exhibit the *right behavior*, and for the *right reasons***,** before firm conclusions can be drawn from the simulation. Nevertheless, the simulation model presented in this paper provides tentative support for the causal hypothesis presented earlier. We therefore present the findings below as preliminary.

Preliminary Finding 1: The accumulation of undiscovered rework from the science project creates a tipping point during production development.

A tipping point is a threshold condition that, when crossed, shifts the dominance of the feedback loops controlling a process [Sterman 2000]. In our case, the shift is from the dominance of the balancing loop *B5: Assured Quality and Release* to the reinforcing loop *R4: Schedule Pressure*

Magnified Ripple Effect. When B5 dominates, progress on the project is made, however steadily, towards completion. When R4 dominates, progress nearly halts as the ripple effect causes the generation of more and more rework.

Preliminary Finding 2: Applying high pressure, or even moderate pressure for extended periods, can push the program past the tipping point.

The tipping point dynamic is clearly seen in the simulation graph presented in Figure 8.[14] The degree to which schedule pressure is applied to workers is measured on a six-point scale: None, Low, Medium-Low, Medium, Medium-High, or High. During times of high schedule pressure, applying the full pressure to the workers might mean that the workers' jobs would be in jeopardy if they did not put forth their full effort, whereas medium pressure might mean that their performance evaluations would be significantly impacted. The graph shows that applying pressure at Medium level and above pushes the discovery (and generation) of quality issues over the tipping point towards more and more rework (this is where loop R4 dominates). Applying pressure at a Medium-Low level and below results in completion of the project (this is where loop B5 dominates).[15]

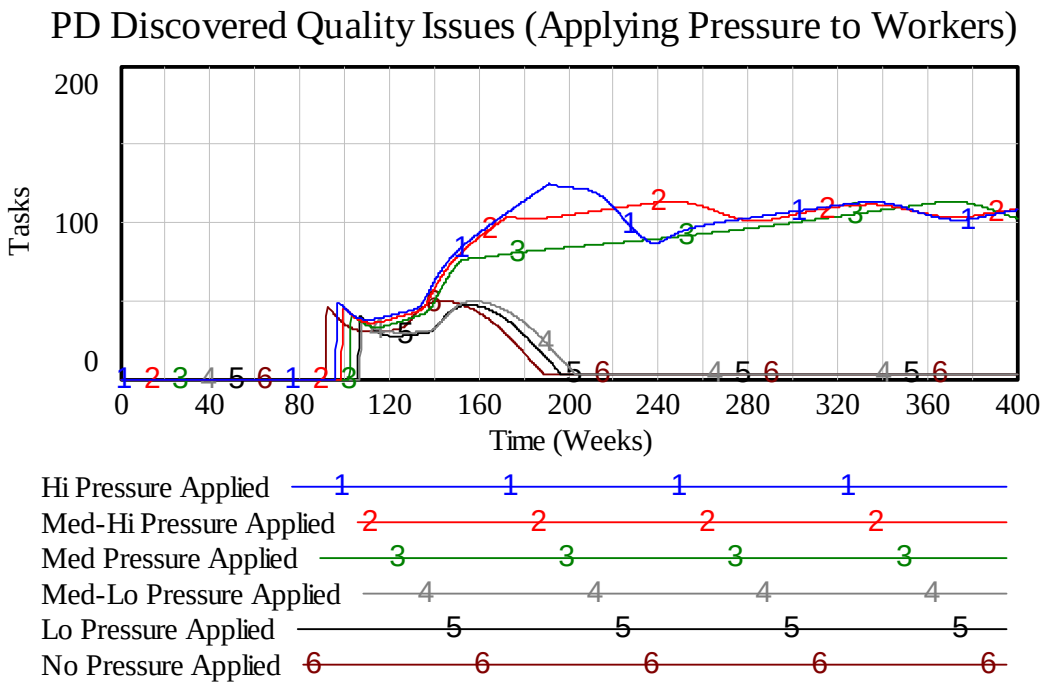


Figure 8: Simulation Results for Production Development Discovered Quality Issues

[14] This and subsequent graphs were generated using the Vensim modeling and simulation tool. These are all behavior-over-time graphs and, as such, the X-axis for these graphs is specified in weeks (400 weeks is the duration of the simulation). Each simulation run is specified as individual graphs distinguished both in color and with a number label (1 through 6 in Figure 8), as specified in the legend below the graph. Each simulation run varies a key parameter. The graph in Figure 8 varies the extent of Applying Pressure to Workers from no pressure applied up to high pressure applied, and shows the value of PD Discovered Quality issues along the Y-axis for all times during the simulation. The label on the Y-axis indicates the units for the variable being graphed, which in this case is the number of tasks involved with fixing the quality issues that were discovered.

[15] The simulation runs for PD Discovered Quality Issues at the lower pressure levels (runs 4, 5, and 6) shown in the figure do not actually show the levels going to zero in the second half of the simulation. The project does terminate, but some minimal level of rework is allowed to remain even at project conclusion, i.e., not all loose ends are completely tied up.

The above behavior is even worse than what we expected as described in the causal hypothesis and as shown in Figure 2. Not only does the ripple effect delay the completion of the project, but in the worst case it extends it out indefinitely toward more and more rework. Of course, if the project were to restart itself on solid ground it could again begin to make solid progress toward definite completion. However, the simulation suggests that if the project were left to run its course, no conclusion would be reached, indicating that the project was past the tipping point. Notice also the increasing amplitude of the (damped) oscillations apparent when higher levels of pressure are applied. These oscillations in the model simulation occur precisely because of the higher levels of undiscovered rework transferred to the production development, as seen below in Figure 9.

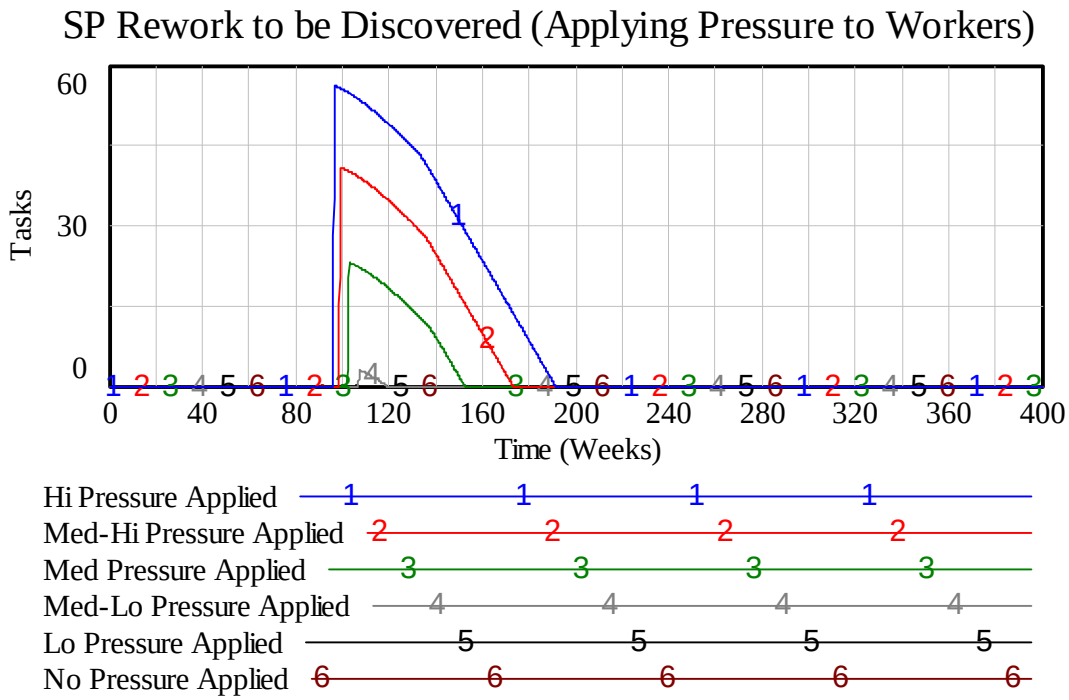


Figure 9: Simulation Results for Science Project Rework to be Discovered

The reason that all runs have no discovered quality issues up until between week 80 and week 120 is that this is the point at which the science project ends and the quality issues from the prototype are transferred to the production development. The graphs in Appendix F show the behavior of the science project prototype development prior to the point of this transfer for the case in which all pressure is applied. Key to the buildup of the initial PD Discovered Quality Issues is the science project's response to the schedule pressure by reducing the rigor of QA processes, and the resulting poor quality work that the schedule and feature-driven development entails.[16]

Preliminary Finding 3: The tipping point contributes to the "90% Syndrome."

[16] Appendix G presents other simulation behaviors at varying levels of pressure applied to workers.

As described in Figure 3 of the causal hypothesis, the model simulation does reflect the 90% Syndrome. Surprisingly, as shown in Figure 10, the only place this phenomenon is seen is where the project has gone past the tipping point, i.e., where above moderate pressure is applied to workers to meet schedule demands in simulation runs 4, 5, and 6.[17] It is significant that the simulation's behavior in seeing forward progress plateauing well before project completion is very consistent with the predictions of the 90% Syndrome. The fact that the simulation plateaus near the 80% level may be due to the fact that the model takes into account all rework remaining in its calculations.

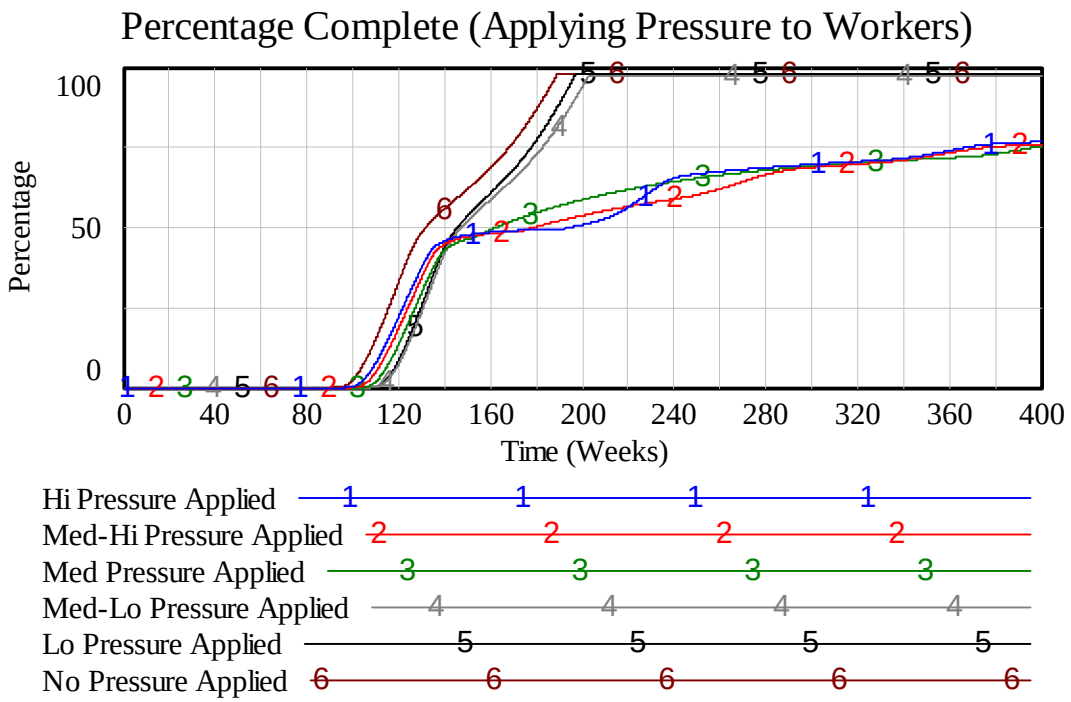


Figure 10: Simulation Results for Percentage Complete

One unexpected observation of the above dynamic is that it appears the simulation requires going over the tipping point before the project displays the 90% syndrome behavior. Progress on the projects that do complete (i.e., simulation runs 4 through 6 in Figure 10) appears to increase in the final weeks. It isn't clear whether actual acquisition programs that exhibit the 90% syndrome are necessarily "over the tipping point" for that project. It may be unlikely, but if true it would be informative about what needs to be done in those circumstances (e.g., letting the project run its course might not be the wisest option, as it would never complete).

[17] The simulation runs for Percentage Complete at the lower pressure levels (runs 4, 5, and 6) shown in the figure do not actually show 100% completion in the second half of the simulation. The project does terminate, but some minimal level of rework is allowed to remain even at project conclusion—i.e., not all loose ends are completely tied up.

Preliminary Finding 4: Applying only a low level of pressure to workers for a limited time, even in the face of heavy schedule demands, shortens project duration through increased productivity.

The above simulation graphs might seem to suggest that the less pressure applied to workers, the better. However, a Vensim™ [18] optimization shows that a steady, low level of pressure is optimal for reducing project duration. While this has not been tested explicitly as part of the model, it may be that by allowing periods of heightened pressure on workers, followed by periods of relaxation, the program could limit the potential for worker burnout, and perform better regarding schedule. This may be a point of future investigation.

Preliminary Finding 5: From a technical standpoint, an earlier transition from the science project to production development creates fewer problems than a later transition does.

Figure 11 shows the accumulation of PD Discovered Quality Issues when the scope of the SP effort is varied. We measure the scope of the SP effort based on the number of features developed, but show it qualitatively in the graph on a four-point scale: Low, Medium-Low, Medium, or High. A medium SP scope was the assumed level in all previous simulations represented above. To simplify the analysis we assume that all schedule pressure is applied to the workers as the science project develops the prototype, so the simulation goes past the tipping point in the case of a medium-feature scope. As is apparent from the figure, going to a high-feature scope (in simulation run 2) causes a later start of the production development, again surpasses the tipping point, and results in higher amplitude oscillations. Reducing the scope to medium-low (in run 3) is still beyond the tipping point, but results in a significantly damped oscillation. Finally, reducing the scope of the development to a low level (simulation run 4) results in successful completion of the production development at approximately week 140, without the project passing the tipping point. The only difference between this run which successfully completed and the runs that passed the tipping point was the scope of the science project prototype. Clearly a science project prototype needs to demonstrate viability and utility, but this simulation shows that going beyond that can lead to disastrous consequences for the project as a whole.[19]

[18] Vensim is the commercial product that was used to develop the causal loop diagrams and the system dynamics model described in this report, as well as to generate the Behavior Over Time (BOT) graphs of the simulation results. Vensim is a registered trademark of Ventana Systems, Inc.

[19] Appendix H provides other simulation behaviors for various levels of prototype scoping.

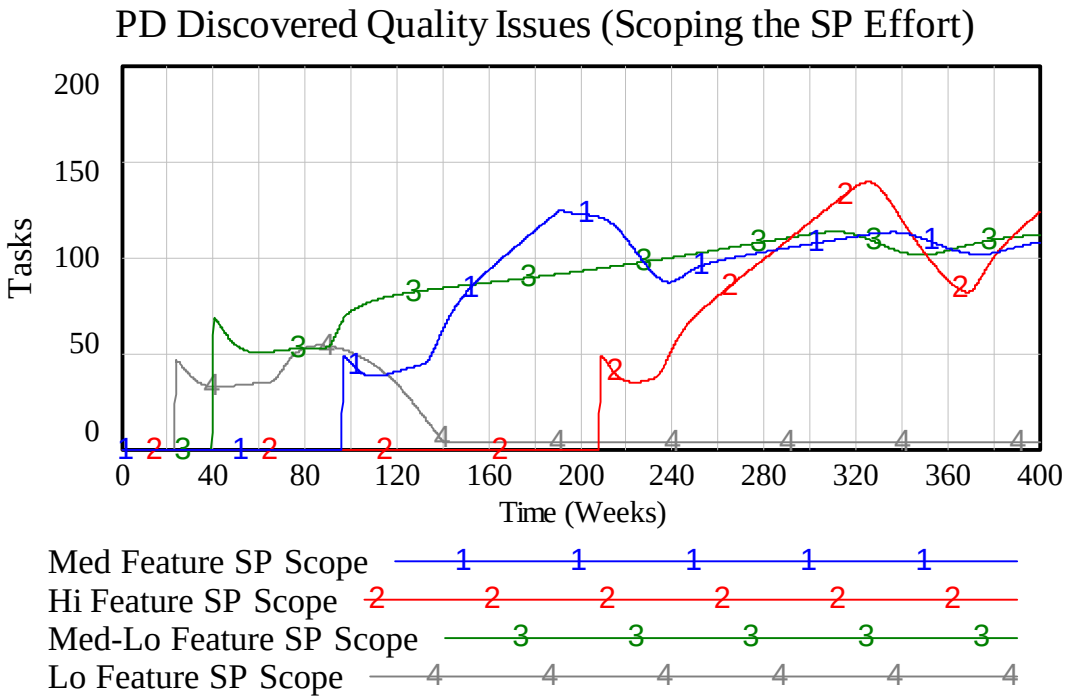


Figure 11: Simulation Results for Production Development Discovered Quality Issues

Preliminary Finding 6: The project performs significantly better by shelving the prototype when it is complete, instead of reusing a poorly-constructed prototype

Not unexpectedly, the simulation predicts a better outcome for shelving the prototype before moving to production development. It is difficult to envision a worse outcome than a project that goes over the tipping point, spiraling downward towards more and more rework. Fortunately, shelving the prototype and starting over in production development does not also go over the tipping point—it was the decision to reuse the poorly-constructed prototype that caused the failure. Figure 12 shows that not reusing the prototype in simulation run 2 leads to project completion around week 180, whereas reusing the prototype leads to ever increasing estimated completion dates. The primary reason is the reduction in the amount of rework the production development has to deal with as shown in Figure 13.

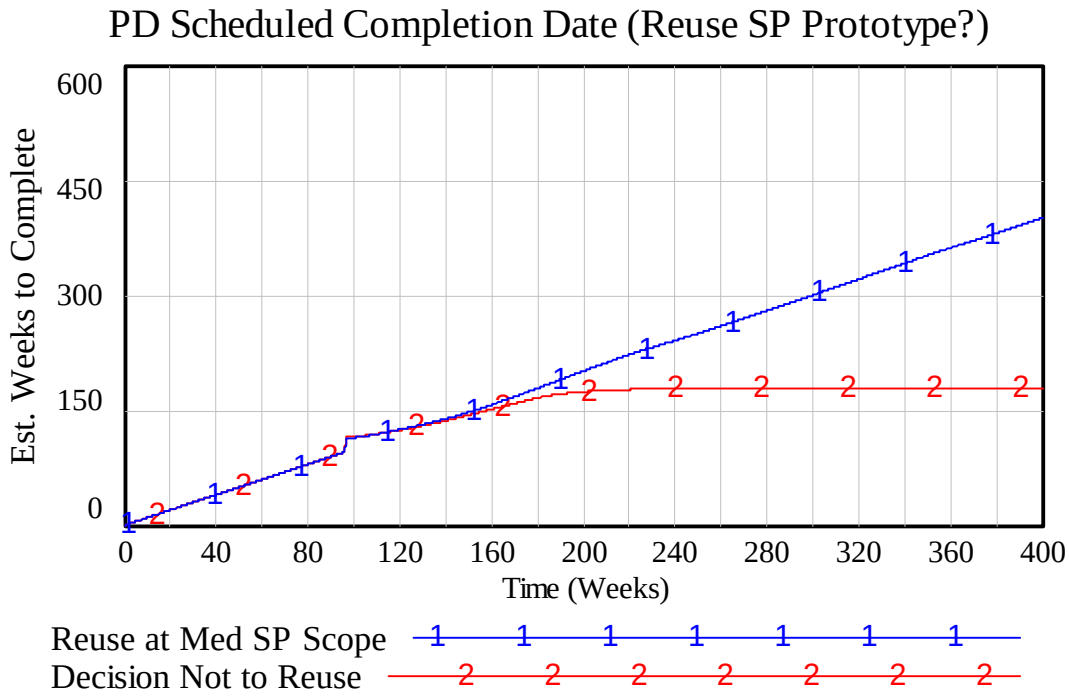


Figure 12: Simulation Results for Production Development Scheduled Completion Date Reflecting the Decision to Reuse

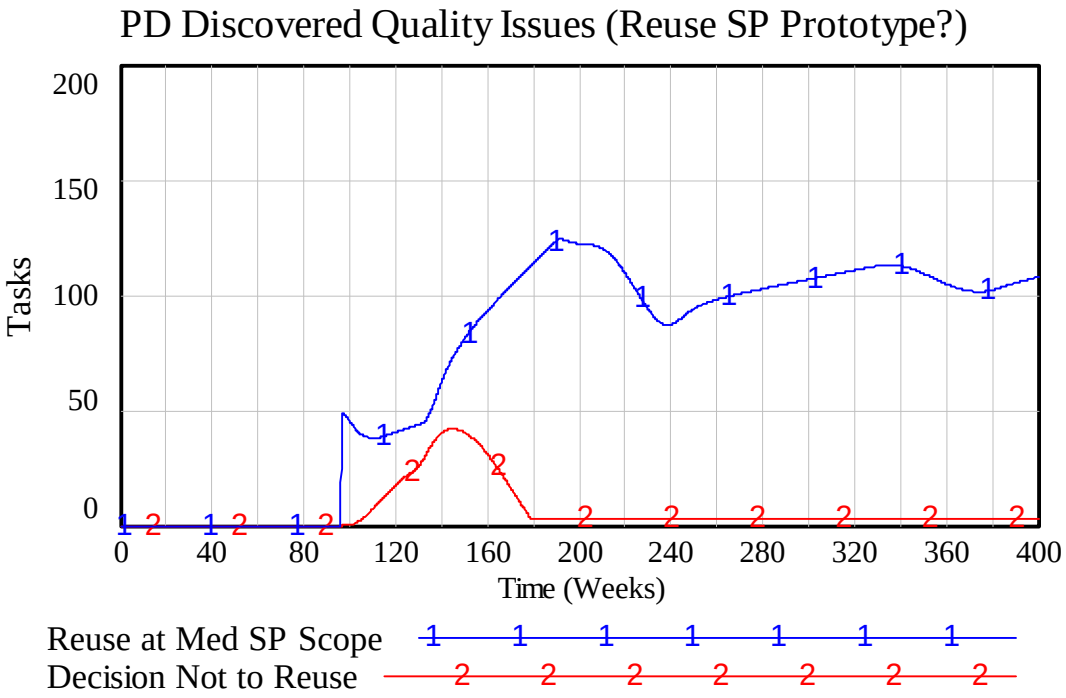


Figure 13: Simulation Results for Production Development Discovered Quality Issues Reflecting the Decision to Reuse

That said, it was not strictly the decision to reuse the prototype that created the disastrous results. Figure 14 below shows that reusing better-constructed prototypes results in approximately the same project duration. In other words, the benefits of doing a science project can be achieved while still keeping to a reasonable schedule, as long as the quality of the prototype does not slip. Interestingly, the best project performance observed was where the science project prototype scope was low, in which case the production development reused the prototype to finish the project in under 150 weeks.

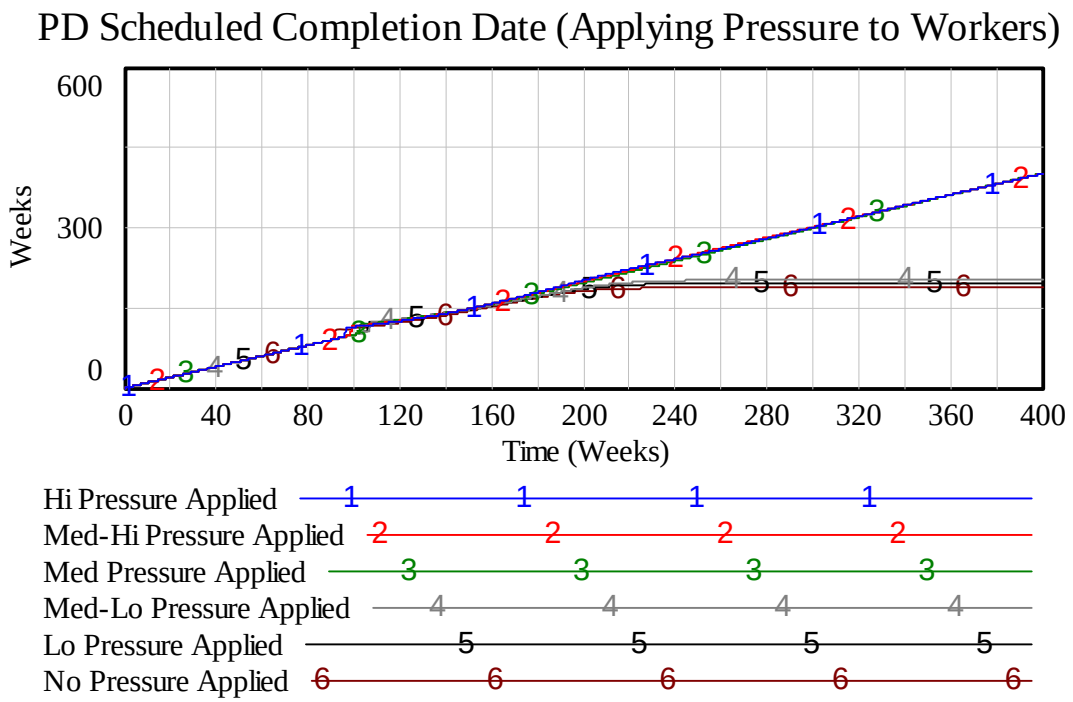


Figure 14: Simulation Results for Production Development Scheduled Completion Date for Various Worker Pressures

A key conclusion from this investigation is that the adverse consequences of "The Evolution of a Science Project" dynamic will take hold if the project develops a "throwaway" prototype and nonetheless attempts to evolve it into a production-quality system. The project will likely also slip schedule significantly if it stops development, discards the "throwaway prototype," and redesigns/re-implements the prototype from the beginning (perhaps incorporating the lessons learned from the prototype implementation)—but it will likely still complete development of the system. If those were the only two options, the question would be one of when to discard the prototype and shift over to conducting a formal acquisition program. A third possible option may be to deploy the last viable version of the prototype, stop development on the prototype, stop using it for further new development, and use the final prototype release as a way to "buy some time" with users—while the development team begins the formal reimplementation effort. However, this approach also has a drawback in terms of the prototype maintenance; when the prototype system does fail in a significant way in the field, someone will likely have to be assigned to understand

what went wrong, and how to work around the problem, consuming more of the limited project resources.

However, experience suggests that the path that is most likely to be most successful in a science project context is to develop an *evolutionary* prototype so that the project will neither "hit the wall" when development stalls, nor "go dark" when the prototype must be discarded and re-implemented. Because there is less build-up of undiscovered rework, and no time-consuming re-start and re-implementation, development can continue through to completion at a reasonable pace. The problem, though, originates with the fundamental conditions and incentives that tend to drive the creation of science projects. If the development effort is led by operational, functional, or business people, their lack of software expertise may make them unaware of this approach, and thus preclude them from using it. Even if the evolutionary prototyping approach were used, it would likely also cost more, and delay the system's first deployment to the field. Neither the time to develop a quality system architecture and design, or produce comprehensive documentation, will be small.

5 System Dynamics Modeling Lessons Learned

This section describes lessons learned about the system dynamics modeling process and the use of modeling tools during the course of this system dynamics development effort. As such these lessons are largely independent of the software acquisition domain, and are relevant to system dynamics modeling efforts in general. Lessons learned are identified in three areas: tooling, the modeling process, and using the model.

5.1 Tooling

Lack of Modularization in Modeling Tools

One significant shortfall of some of the system dynamics modeling tools, including Vensim, is the almost complete absence of the key software engineering concepts of modularity, information hiding, and data abstraction. System dynamics models created in Vensim can only be created as monolithic entities in which all pieces exist at the same level. As a result, it isn't possible to have a node at one level be broken down into a more complex structure at a lower level, and then have those structures be further broken down at even lower levels. Such a capability would allow the true complexity of the model to be hidden at the highest levels, but still be available for inspection and manipulation at the lower levels. While it is possible to "hide" elements of the model from sight in a given user-defined "view" of the model in Vensim, this does not give a logical, multi-level partitioned structure to the model. Instead, it simply allows portions of the model to be viewed independently, even though they are all connected at the same level.

5.2 Modeling Process

Selecting a Worthwhile Dynamic to Model Needs Careful Consideration

An important and difficult aspect of developing a useful system dynamics model is in the initial selection of the dynamic that is being modeled. Identifying concepts for dynamics to be modeled that are at the same time relevant (to the business), important (with a substantial value at risk), compelling (to industry practitioners), and counter-intuitive (compared to everyday behavior, and thus worthy of understanding) is a difficult and time-consuming process. There is little point in investing the time and effort needed to do system dynamics modeling in a concept that is unlikely to provide insights that are equal to that investment. It is this combination of attributes that results in the most interesting concepts, and those most worthy of being modeled.

Both Modeling Experts and Domain Experts are Essential to Modeling

Successfully creating a system dynamics model that provides a realistic depiction of real-world behaviors requires a combination of both domain expertise (i.e., experience with, and knowledge of the domain being modeled) and system dynamics modeling expertise (i.e., experience with systems dynamics modeling using the specific tools required). These areas of expertise rarely overlap, and this is as it should be. This separation helps to prevent the ease or difficulty of modeling some aspect of the problem from influencing how, or whether, that aspect is modeled. The result is, ideally, a model that is a more accurate reflection of reality.

Building a System Dynamics Model Takes Time, and Must be Incremental

It can take a significant amount of time to build a good system dynamics model, and this must be done gradually. It's very important to build the system dynamics model incrementally for several reasons. First, if the model is built quickly, it will be difficult to make it work, as there will be errors present in multiple parts of the model, creating a "big bang" effect which will make it difficult to discover all of the errors. Second, if it's created in this way, it may be difficult to come to a full understanding of the model's behaviors. If it's not possible to explain *why* the model does what it does, then people are unlikely to accept that the model is an accurate representation of the real system's behavior.

System Dynamics Modeling *is* Complex Software Development

The complexity of building a complex system dynamics model is directly comparable to the complexity of software development. While it appears, superficially, that the modeler is only creating a large diagram, this is misleading. Within each symbol in the model is embedded either an equation or a segment of logic, with the same level of strongly typed variables that a standard high-level programming language would require. The use of the diagram produces an appearance of simplicity that is belied by the cumulative complexity of the interactions of all of the diagram's nodes. When completed, a functioning system dynamics model has all of the complexity of an advanced piece of software. As a result, the model must be documented to the same degree (or greater) as would a similarly sized piece of software, if others will be expected to either understand, or enhance it.

Granularity Matters— While All Models are Wrong, Some Models are Useful

It is inevitable that a model of any aspect of the real world is going to be inadequate in a multitude of different ways, primarily because it cannot represent all of the considerations and detail involved. It's not possible, and certainly not practically feasible, for the model to be completely faithful to reality. It will fall short of this goal in some (and likely many) respects, because there will never be adequate time to model the known potential influences in full detail. This means that the choices made by the modelers regarding the level of detail, or granularity of the model, must be very carefully considered. Given this situation, there are legitimate questions about the ultimate usefulness of any model, and whether with its chosen level of abstraction it can still produce valuable insights into real-world behavior. The quote by George Box stating "All models are wrong, but some are useful"[20] sums up this situation well. Our belief is that despite the fact that models such as the one described here are known to be flawed, they still not only produce important insights, but raise valuable questions as well. It is essential to make every effort to understand and validate the structure and behavior of a model—and having done so, we can use the model to create a theory of the system's behavior, and thus predict possible outcomes. When that model and theory are found to be incomplete or inaccurate, we will need to change or extend the model to address those additional issues, and further evolve our theory. Until that time, we believe that even a known flawed model (as all models inevitably are) will be significantly more useful in understanding complex behaviors than having no model at all.

[20] George E. P. Box, "Robustness in the Strategy of Scientific Model Building," in Robustness in Statistics: Proceedings of a Workshop edited by R.L. Launer and G.N. Wilkinson, May 1979.

Scoping Matters— Choosing How Much Modeling is Good Enough

The early question of deciding what scope to include in a model is essential. The only realistic answer is to say that it depends on how the model will be used. If the model is attempting to illustrate a notional concept of the scenario for educational purposes, a significantly narrower scope or view of that aspect of the world is needed than would be the case for, say, a model which will be used to try to predict the specific ultimate outcomes of a complex system (i.e., when the program will be completed, and how much it will cost).

The "I Already Knew That" Effect

One seeming paradox faced by the developers of system dynamics models is that, when the models have been carefully constructed to be representative of a given real-life dynamic behavior, the reaction of some upon seeing the results can be, "I already knew that!" In short, after the model has been completed, the behavior it demonstrates may appear to some as being obvious or trivial. This can occur because while the result might not be obvious to a novice in the field of study, the deeper understanding possessed by an expert or veteran professional may well include an intuitive or notional understanding of the dynamic. With sufficient real-world experience people often have many general notions of how things relate to, and interact with one another, and what the outcomes of certain actions might be—but most would be hard-pressed to commit to a firm *prediction* of the future behavior of any complex system given a set of specific inputs. An approach such as system dynamics can provide more detailed and quantitative information about the dynamics, as well as a solid foundation for developing practical mitigation approaches—something that is difficult to do with only a simplified, qualitative understanding of the behaviors. The reality is that "hindsight is 20/20," and it is easier to confirm one's belief in the likelihood of an event that has already occurred, than it is to make that same claim beforehand. Furthermore, even if the results of simulating a model do not produce a profound new insight into a situation, the fact that an intuition has been validated as correct by the simulation's results is still important and significant.

One Man's Meat is Another Man's Poison

In some cases it may appear that the results produced by a model appear to be uninteresting or even trivial in the eyes of a domain expert. However, those same results may be unexpected and insightful to a practitioner, or to a student with less experience. In this situation it is the difference in perspective that is significant. The discovery of a new insight, even if it is already well understood by some (even to the point of being a proverb), is no less profound or valuable to the person making that discovery. If it does appear to be trivial to the experts, the following question becomes relevant: "If this is so simplistic, and has been seen so many times before, then why hasn't it been resolved?" If the behavior is truly that simple, then why hasn't it gone away? Why is it still plaguing us?

5.3 Using the Model

Tipping Points Provide Critical Insights

One important value of creating and studying a system dynamics model is identifying inherent "tipping points" in the model—places where the output behavior of the model may change suddenly and drastically, even though the inputs are changing steadily and smoothly. These dramatic shifts, sometimes the result of shifting loop dominance, can occur with little prior warning. Such

tipping points can be relatively common, especially in system dynamics models that exhibit feedback and nonlinear behavior. Tipping points can be disturbing both because of their effect on the successful operation of the system, and their seeming unpredictability. Tipping points can be passed with few outward signs, like a boat sailing over a tsunami in the deep ocean, which will barely notice the seemingly small wave passing underneath it. It's only when the tsunami eventually rises out of the shallow water of the coast that it becomes destructive.

Communicating Complex Models is Difficult

One potentially valuable use of a model is in communicating some of the model's key behaviors to others. This can be difficult when working with a system dynamics model, as the model will typically feature more complexity than can be grasped by even the most interested observer without spending a great deal of time studying it. Thus, for a system dynamics model to be used for communicating and explaining important behaviors, it must be substantially simplified. This is often done by creating a very high-level causal loop diagram which features only the essential loops from the model. While time-consuming, this approach is the only viable way to make the model comprehensible. The loops featured in this top-level diagram should be selected carefully, resulting in the minimum set of loops that can explain the overall behavior of the system.

6 Summary, Conclusions, and Next Steps

Software acquisition programs regularly experience recurring cost, schedule, and quality failures, and progress and outcomes often appear to be unpredictable and unmanageable. The subject of this report is a specific adverse acquisition dynamic concerning the poorly controlled evolution of small prototype efforts into full-scale systems that is called "The Evolution of a Science Project." This report provides a narrative from an actual acquisition program that exemplifies the dynamic, qualitatively describes its key aspects, and presents some of the most relevant prior research done on one of those aspects, project rework. The report describes a system dynamics model that was built to analyze this dynamic, and presents six preliminary findings:

- The accumulation of undiscovered rework from the science project creates a tipping point during production development (*at the tipping point program behavior changes unexpectedly from completing in a finite time, to not completing at all*).
- Applying high pressure to developers, or even moderate pressure for extended periods, can push the program past the tipping point.
- The tipping point contributes to the 90% Syndrome (this occurs when development stalls near completion, with little progress made despite spending more time and effort).
- Applying only a low level of pressure to workers for a limited time, even in the face of heavy schedule demands, shortens project duration through increased productivity.
- From a technical standpoint, an earlier transition from the science project to production development creates fewer problems than a later transition does.
- The project performs significantly better by shelving the prototype when it is complete, instead of reusing a poorly constructed prototype.

The last section presented general lessons learned about the system dynamics modeling process and tools. We also make six observations about the activity of system dynamics modeling of software-reliant acquisition that are drawn from our experiences both with this effort, and with software-reliant acquisition programs in general:

- Many software-reliant acquisition program behaviors are an interaction between the inherent structural dynamics (i.e., the "physics") of the software development activities, and the incentives that are acting upon the key participants and stakeholders as they make decisions.
- System dynamics modeling can provide a deep understanding of the often poorly understood dynamics of software-reliant acquisition.
- It's helpful to have a clear understanding of a given dynamic before selecting the most appropriate course of action for a given program (so that the action addresses the actual problem, rather than only a symptom).
- Tipping points may play a key role in many complex and challenging software development and acquisition situations, and require further study.

- Building models that are valid and reliable is a significant undertaking, requiring significant effort, as well as both detailed domain expertise and extensive modeling experience.
- There is much more that can be learned about the problematic behaviors of software-reliant acquisition programs and their potential solutions by applying system dynamics modeling to other aspects of software-reliant acquisition.

Each of these conclusions has been discussed at some length in other parts of this report, and so need no further clarification here. However, there is also an overall conclusion that can be drawn from these six, and it is worthy of emphasis—that *modeling the incentives and dynamic behavior of acquisition programs is one of the most powerful and cost-effective ways available of understanding problems, evaluating candidate solutions, and producing better acquisition program outcomes*. It is neither cost-effective nor practical to conduct large-scale experiments on acquisition programs of record, which makes modeling these systems the most viable method for studying them and analyzing the comparative advantages of different practices and methods. Even the most minor efficiencies gained in large acquisition programs from using modeling could produce benefits in functionality, quality, performance, improved time-to-deployment, and direct cost savings that would recoup the research and analysis investment many times over. We hope to continue to have the opportunity in future work to prove this to be the case.

Future Directions

There are a number of enhancements to the system dynamics model of "The Evolution of a Science Project" that have been proposed, but which could not be incorporated in the course of the initial year's work. These are described in Appendix J.

There are also many different research directions which would be useful to pursue in moving forward with the modeling of software-reliant acquisition dynamics. A few of these are discussed in the following sections.

Modeling Potential Mitigations and Solutions

An important next step in this work is to mitigate the effects of misaligned acquisition program organizational incentives and adverse software-reliant acquisition structural dynamics by improving program staff decision-making. This requires not only modeling and analyzing adverse acquisition dynamics that have been encountered in actual programs, but identifying and then modeling candidate solutions that may be able to mitigate or resolve the counter-productive behaviors. Constructing potential solutions and then connecting them to the original model of the program behavior makes it possible to evaluate the ability of each candidate approach to mitigate the dynamic's adverse behavior. This allows an initial assessment of the value of the approach to be made, which could then be initially piloted on a small program to see how well the predicted model behavior mapped to the real world.

Interactive Learning Exercises

Education in a classroom setting does not always translate to the reality of complex challenges and changing priorities students face in an operational environment—and as a result, may not produce significant job performance improvement. While classes may focus on the facts of a top-

ic, they may fall short in teaching what is normally acquired through experience—that is, the types of problems that are most commonly encountered, and ways of avoiding these issues.

The research evidence indicates that "mastery experiences"—actual successes at dealing with difficult situations—most strongly promote effectiveness in an activity, making them more valuable for learning, developing the skills, critical attitudes, and confidence to function successfully in an operational environment [Bandura 1986]. Creating interactive experiential learning tools such as "flight simulators" for use in the classroom is one possible avenue for providing students with mastery experiences. Compared to traditional education, they can produce better understanding and longer retention through an active learning experience. Flight simulators have been used for years to give pilots experiential training in flying aircraft, especially in conditions that would be impossible or too dangerous to test with real aircraft. Similarly, management flight simulators are just as important for providing education in complex system management domains—both technical and social—efficiently and effectively.

System dynamics models provide an opportunity to give students this kind of experiential learning by allowing them to interact with the dynamics of the kind of problems they may face in an operational environment. The system dynamics model developed for "The Evolution of a Science Project" dynamic can provide the foundation for an interactive learning simulation. Some of the key considerations in creating this type of simulation tool include a) defining the proper learning objective(s) of the simulation, b) the types and degree of control the student is offered, and c) the difficulty in succeeding at the simulation. Having a clear learning objective (or small set of objectives) is important both in focusing the exercise for the student, as well as in making the development effort manageable. The user inputs and controls should be kept to a minimum, focused almost exclusively on the key aspects of the scenario that have the greatest influence on the simulation. The difficulty in succeeding at the simulated scenario should be enough to test even advanced students, and ideally will present them with counter-intuitive problems that will challenge their preconceived mental models of the scenario. The chance to give students an important insight into a real-world problem by having them discover it, rather than by being told, is especially valuable.

Some examples of educational simulation tools relating to software development and project management include the SimSE Software Engineering Simulation Environment (SimSE), [Navarro 2006][21] and the pmBLOX project management tool.[22]

The experience of using interactive simulations to learn about acquisition helps significantly in both analyzing and making decisions about these complex situations. By improving program decision-making, we can help programs overcome counter-productive behaviors that stem from misaligned incentives and underlying dynamics, and thus deploy higher-quality systems to the field in a more timely and cost-effective manner.

[21] Additional information on SimSE can be found at http://www.ics.uci.edu/~emilyo/SimSE

[22] Additional information on pmBLOX is available at http://www.tucows.com/preview/853935

Agent-Based Modeling

System dynamics modeling is by no means the only approach that can be used to model complex systems such as acquisition programs. Another widely-used approach is agent-based modeling. The difference between the two is that system dynamics modeling and agent-based modeling view problems from different perspectives. System dynamics modeling

- uses a top-down approach based on historical behaviors
- models the causes that underlie the problematic behavior
- helps understand the reasons why problematic behaviors arise

In contrast, agent-based modeling

- uses a bottom-up approach based on “rules” that are followed by each agent
- models behaviors of individual agents that are autonomously interacting with others
- helps understand emergent behavior based on local agent interactions

System dynamics modeling aggregates behavior at a high level in order to control the model’s complexity. Agent-based modeling looks at problems in terms of individual low-level actors. It supports the analysis of the “emergent behavior” of those agents each acting according to a defined set of rules.

In terms of selecting the most appropriate approach to use for a given situation, agent-based modeling is commonly used when behavior can’t be aggregated, such as in analyzing decision-making. Agent-based modeling is generally most appropriate when

- the focus is on the behavior of interacting autonomous agents
- agents have to behave realistically as their real-world counterparts (i.e., persons, organizational entities, etc.) would
- agents adapt both to the environment, and to other agents' actions
- agents act strategically based on the likely actions of other agents
- modeling cooperation
- future actions may not be the direct consequence of past actions
- the system's complexity is beyond the modeling capabilities of other approaches [Siebers 2010]

Practically speaking, many complex systems such as acquisition will exhibit a combination of behaviors, some of which are best suited for system dynamics, and others which are most appropriate for agent-based modeling. Hybrid modeling approaches are becoming more common as a result of this. It is likely that modeling software-reliant acquisition programs would be well served by such a hybrid approach, given the combination of process execution and individual decision-making that is involved.

7 References/Bibliography

URLs are valid as of the publication date of this document.

[Abdel-Hamid 1991]
Abdel-Hamid, Tarek & Madnick, Stuart. *Software Project Dynamics: An Integrated Approach*. Prentice Hall, 1991 (ISBN 978-0138220402).

[Adams 2004]
Abelson, Linda A.; Adams, Richard J.; & Eslinger, Suellen. "Acquisition Modeling: The Key to Managing Acquisition Complexity?" *Proceedings of the Third Annual Conference on Software-Intensive Systems Acquisition*, January 2004.

[Bandura 1986]
Bandura, Albert. *Social Foundations of Thought and Action: A Social Cognitive Theory*. Prentice Hall, 1986 (ISBN 978-0138156145).

[Brougham 1999]
Brougham, William. "System Dynamics and Process Improvement: Can the U.S. Navy Acquisition Community Learn from Industry Behavior?" MS diss., Alfred P. Sloan School of Management, Cambridge MA, Center for Information Systems Research, 1999. http://handle.dtic.mil/100.2/ADA372210

[Buettner 2008]
Buettner, Douglas John. "Designing an Optimal Software Intensive System Acquisition: A Game Theoretic Approach." PhD diss., University of Southern California, 2008. digitallibrary.usc.edu/assetserver/controller/item/etd-Buettner-2432.pdf

[Choi 2006]
Choi, KeungSik & Bae, Doo-Hwan. "Analysis of Software-Intensive System Acquisition Using Hybrid Software Process Simulation," 254-261. *Software Process Change, Lecture Notes in Computer Science*, Volume 3966. Springer-Verlag, 2006 (ISBN 978-3540341994).

[Christie Raffo 1998]
Raffo, David M., Vandeville, Joseph V., & Martin, Robert H. "Software Process Simulation to Achieve Higher CMM Levels." *Journal of Systems and Software 46* 2/3 *(*April 1999).

[Cooper 1980]
Cooper, Kenneth G. "Naval Ship Production: A Claim Settled and a Framework Built." *Interfaces 10*, 6 (December 1980): 20-36.

[Crinnion 1991]
Crinnion, John. *Evolutionary Systems Development: A Practical Guide to the use of Prototyping Within a Structured Systems Methodology*. Plenum Press, 1991.

[Ford 1998]
Ford, David N. & Sterman, John D., "Dynamic Modeling of Product Development Processes," *System Dynamics Review 14,* 1: 31-68. 1998.

[Ford 2007]
Ford, David N., Lyneis, James M.,& Taylor, Timothy R.B., "Project Controls to Minimize Cost and Schedule Overruns: A Model, Research Agenda, and Initial Results." *Proceedings of the 2007 International Conference of the System Dynamics Society*. Boston, MA, July 2007.

[GAO 2006]
DoD Acquisition Outcomes: A Case for Change. Statement of Katherine V. Schinasi, Managing Director, Acquisition and Sourcing Management, United States Government Accountability Office, GAO-06-257T, November 15, 2005.

[Haberlein 2004]
Haberlein, Tobias. "Common Structures in System Dynamics Models of Software Acquisition Projects." *Software Process Improvement and Practice* (September 2004): 67–80.

[Homer 1993]
Homer, J. & Sterman, J. "Delivery Time Reduction in Pump and Paper Mill Construction Projects: A Dynamic Analysis of Alternatives*." Proceedings of the 1993 International System Dynamics Conference*. Monterey Institute of Technology, Cancun, Mexico. 1993.

[Kadish 2006]
Kadish, Ronald. *Defense Acquisition Performance Assessment Report.* Assessment Panel of the Defense Acquisition Performance Assessment Project, January 2006. https://acc.dau.mil/CommunityBrowser.aspx?id=18554

[Li 2011a]
Li, Ying. "The Impact of Design Rework on Construction Project Performance," *Proceedings of the 29th Annual International Conference of the System Dynamics Society*, Washington D.C., July 2011.

[Li 2011b]
Li, Jiang, Taylor, Timothy R.B., & Ford, David N.. "Impacts of Project Controls on Tipping Point Dynamics in Construction Projects," *Proceedings of the 29th Annual International Conference of the System Dynamics Society*, Washington D.C., July 2011.

[Lyneis 2007]
Lyneis, James M. and Ford, David N., "System Dynamics Applied to Project Management: a Survey, Assessment, and Directions for Future Research," *System Dynamics Review 23* 2/3 (Summer/Fall 2007): 157-189.

[Madachy 2006]
Madachy, Raymond; Boehm, Barry; & Lane, Jo Ann. "Spiral Lifecycle Increment Modeling for New Hybrid Processes," 167-177. *Proceedings of SPW/ProSim'2006.* Shanghai, China, May 2006.

[Madachy 2008]
Madachy, Raymond J. *Software Process Dynamics.* Wiley-IEEE Press, 2008 (ISBN 978-0471274551).

[Navarro 2006]
Navarro, Emily. "SimSE: A Software Engineering Simulation Environment for Software Process Education." PhD diss., Donald Bren School of Information and Computer Sciences, University of California, Irvine, 2006

[Repenning 2001]
Repenning, Nelson P.; Goncalves, Paulo; & Black, Laura J. "Past the Tipping Point: The Persistence of Firefighting in Product Development." *California Management Review* (July 2001).

[Scacchi 1993]
Scacchi, W. and Mi, P. "Modeling, Enacting and Integrating Software Engineering Processes." *Proceedings of the 3rd Irvine Software Symposium*, Costa Mesa, CA, April 1993.

[Scacchi 2001]
Scacchi, Walt & Choi, S. James. "Modeling and Simulating Software Acquisition Process Architectures." *Journal of Systems and Software 59,* 3 (December 2001): 343-354. http://www.ics.uci.edu/~wscacchi/Software-Process/Readings/ProSim-2K-JSS-2001.pdf

[Siebers 2010]
Siebers, P. O., Macal, C.M., Garnett, J., Buxton D., & Pidd, M. "Discrete-Event Simulation is Dead, Long Live Agent-Based Simulation!" *Journal of Simulation 2010*, 4: 204–210.

[Sterman 2000]
Sterman, J. D.. *Business Dynamics: Systems Thinking and Modeling for a Complex World.* McGraw-Hill, 2000.

[Taylor 2005]
Taylor, Tim, Ford, David N., & Johnson, Scott. "Why Good Projects Go Bad: Managing Development Projects Near Tipping Points." *Proceedings of the 23rd International Conference of the System Dynamics Society*. Boston, MA, July 2005.

Appendix A: System Dynamics Method Background

The following sections discuss five seminal efforts that explored modeling of the rework dynamic, three from a general project management perspective, and two from a software development project management view:

- Ken Cooper, *Naval Ship Production: A Claim Settled and a Framework Built* [Cooper 1980]
- Tarek Abdel-Hamid and Stuart Madnick, *Software Project Dynamics* [Abdel-Hamid 1991]
- David Ford and John Sterman, *Dynamic Modeling of Product Development Processes* [Ford 1998]
- Tim Taylor and David Ford, *Why Good Projects Go Bad: Managing Development Projects Near Tipping Points* [Taylor 2005]
- Raymond Madachy, *Software Process Dynamics* [Madachy 2008]

7.1 Cooper

The model development in the context of the Ingalls Shipyard claim [Cooper 1980] continues to be a seminal work in system dynamics modeling of project management.

While Cooper does not describe the structure of the model that was created for this analysis, a key aspect of the model is the "ripple effect" incurred when rework is required due to a design change. These ripple effects are second- and third-order effects stemming from the original change that magnify, or "snowball" due to rescheduling of work, scheduling conflicts that result from the rescheduling, undiscovered work errors that result in additional impacts, and other aspects that add cost and schedule to the original rework effort.

An important difference between Cooper's rework model and other subsequent models is that it does not explicitly track rework that has *not* been identified or discovered, simplifying the model and avoiding that additional delay in performing rework. However, as latent defects requiring rework are still undeniably present in software systems, this simplification may not accurately depict reality.

7.2 Abdel-Hamid and Madnick

Abdel-Hamid and Madnick [Abdel-Hamid 1991] describe a complex process of error detection and rework consisting of the following steps:

- *Error Detection Rate (Potential and Actual)*: The *potential* error detection rate is the theoretical maximum, which depends on the level of quality assurance (QA) spent, and varies based on whether it is a design or coding error (design errors are generated more frequently, and are more expensive to find), and the density of errors (higher densities of errors make them easier to find). The *actual* error detection rate takes into account productivity losses in the QA effort due to communication and overhead. Regardless of the effort spent in QA, it will not be possible to detect all errors.

- *Hierarchy of Errors*: Errors vary from the obvious to the subtle, with the most subtle errors occurring infrequently, but being the most costly to detect. Subtle errors are also discovered later in the development process, when they will be the most costly to rework.
- *Rework (Nominal and Actual)*: The rate of rework depends on the amount of effort spent in rework, and the amount of effort needed to rework each error. Nominal rework depends on the error type (design or coding), and the current development phase of the project. Actual rework is based on the error type as well as the efficiency of the rework staff, which is lessened by productivity losses due to communication and overhead.
- *Bad Fixes*: The rework activity itself introduces errors either by not totally removing the error, or by introducing a new error in the process.
- *Growth of Undiscovered Errors (Active and Passive)*: Requirements and design errors that go undiscovered are considered "active" in that they can cause additional errors in later development phases,[23] in addition to undergoing a steady escalation of the cost to rework them as time passes in the development process. "Passive" errors must also be reworked, but do not produce additional errors (an example might be a simple coding error). No distinction is made between different types of errors beyond classifying them as design vs. coding.
- *Impact of Error Density on Active Error Regeneration*: The escalation in the cost to repair an error is represented through growth in the *number* of errors. The increase in the number of active errors in the model occurs in two ways: (1) the number of new *generations* of "children" errors that an error produces as time passes (depending on its time of discovery—older errors that were introduced earlier in the development lifecycle produce more offspring), and (2) the number of new "children" errors that are produced in each generation. In addition, the multiplier value representing the number of "children" errors increases with the density of active errors.
- *Active Error Retirement*: Active errors continue to produce new errors while development is ongoing, but are "retired" when they stop reproducing toward the end of development and become passive errors. The fraction of active errors being retired increases exponentially as development nears completion.
- *System Testing*: Development resources are gradually shifted to testing as development activities come to a close. Part of the testing effort is spent on detecting and correcting errors. The actual effort needed to test a task is based on the number of errors in that task (which drives the effort spent in detecting and correcting errors), overhead activities performed by the testing staff in developing test plans and test cases, and productivity losses from communication and motivation issues.

[23] As an example, a requirements error will "cause" other errors, as that requirement may have implications on multiple aspects of the system, each of which be designed and implemented differently than it would have been as a result of the original requirements error.

7.3 Ford and Sterman

In the Ford/Sterman model (which has its conceptual basis in Homer and Sterman's model [Homer 1993]), tasks move between several different key stocks[24]: Tasks Not Completed, Tasks Completed but not Checked, Tasks to be Changed, Tasks Approved, and Tasks Released (as shown in Figure 15). Tasks start in the Tasks not Complete stock, and move to the Tasks Completed not Checked stock as they are completed. If no changes (i.e., rework) are required, the tasks are approved and move to the Tasks Approved stock, and are then released into the Tasks Released stock. If changes are discovered by the Quality Assurance (QA) activity, the tasks are added to the Tasks to be Changed stock. The Change Task activity makes the changes and puts them back into the Tasks Completed not Checked stock, where the QA process is repeated.

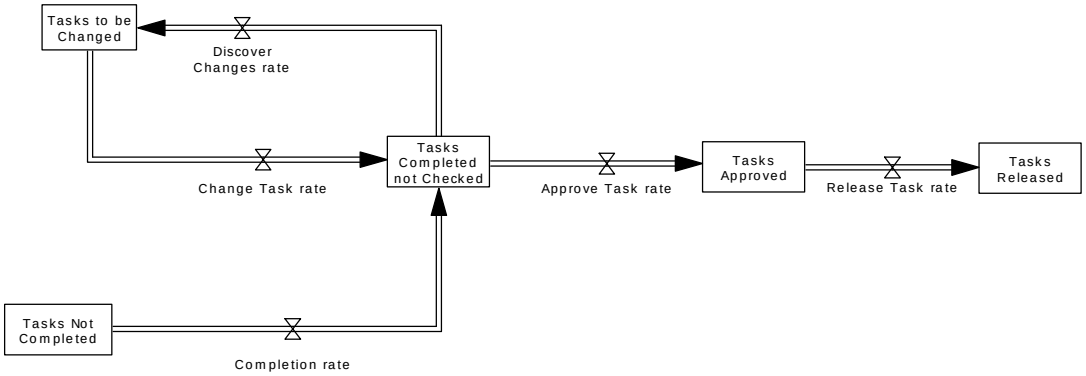


Figure 15: Stock and Flow Diagram of Rework Model [Ford 1998]

7.4 Taylor and Ford

Taylor and Ford describe the behavior of general development projects in the context of nuclear power plant construction using a system dynamics model. To represent the feedback structure involved in rework, the model uses a reinforcing loop connected to a balancing loop. As shown in Figure 16, the balancing loop (B1) describes the flow of work through the Quality Assurance (QA) organization, and on through the approval and release of the completed work, thus reducing the QA backlog. The reinforcing loop (R1) follows the same flow of work through QA, but looks at the discovery by QA of needed rework, leading to a "ripple effect" that adds still more work to the discovered rework because of the impact of those rework changes on other parts of the system, which increases the rework backlog and, after a time delay as the rework is done, the QA backlog. The ripple effect in the model increases as the rate of discovery of rework increases. What can occur over time is that the effect of the balancing loop, namely the approval and release of work, can end up being overcome, or dominated, by the effect of the reinforcing loop that describes the rework. This is referred to as "shifting loop dominance," and the point at which the dominance shifts from the balancing loop to the reinforcing loop is a fundamental change to the dynamics of

[24] The term "stock" refers to a box in a "stock-flow" diagram that represents a collection, or aggregation of items that may accumulate and decline. Correspondingly, a "flow" (often represented as an hourglass or valve) controls the rate at which items are added to, or removed from, a stock.

the system that is referred to as a "tipping point."[25] Degrading projects are those in which current project backlogs exceed previous project backlogs—in other words, those whose backlogs are growing. If current project backlogs are equal to past backlogs (i.e., the project is stagnant, with the backlog neither increasing nor decreasing), the project is still not making any forward progress—and this situation is considered to be an unstable equilibrium, from which the project will move toward either improvement or degradation. Reaching the tipping point in this system is significant, because left unchecked, an infinitely large backlog of rework may ultimately develop, degrading forward progress and leading to project failure.

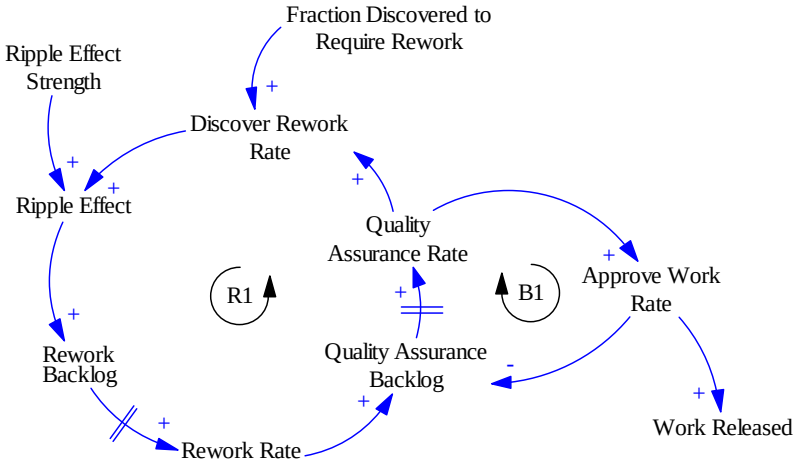


Figure 16: Causal Loop Diagram of Rework Model [Taylor 2005]

7.5 Madachy

Madachy proposes a simple model for rework (based on Abdel-Hamid's earlier integrated project model) in which there is a constant "fraction correct" of task development work being done, with the complementary fraction being done incorrectly ("1 – fraction correct"). After the incorrect task development is discovered at some rate, it is routed back to the task development queue. Madachy uses a different structure for "working off defects" as opposed to performing "task development," in order to take into account a less-than-perfect defect detection process that permits latent defects to continue to exist undiscovered, and to account for and track rework as a separate effort from new development. Madachy refers to this as a "Defect Rework Policy," and describes a "simplistic" model in which all defects are reworked immediately as soon as they are discovered. The resources to do this are assumed to be fully available, and an average delay and effort for fixing each defect is used.

Madachy points out that in practice, it is more likely that there will be delays in performing rework, as it is done based on the relative priorities of the defects that are found, and the staffing constraints for performing rework may need to be modeled as well. Madachy acknowledges that

[25] A tipping point is a threshold that, when crossed, causes a significant change in the output, despite only an incremental change in the input. The concept of a tipping point is further described, and its significance discussed, in further detail in Section 4.3 and Section 5.

these aspects may have significance when trying to develop a more realistic model of the dynamics of rework.

Appendix B: System Dynamics Method Background

Our research uses a technique called system dynamics—a method for modeling and analyzing the holistic nature of complex problems as they evolve over time [Sterman 2000]. System dynamics has been used to gain insight into some of the most challenging strategy questions facing businesses and government for several decades. The Franz Edelman Prize for excellence in management was given in 2001 to a team at General Motors who used system dynamics to develop a successful strategy for launch of the OnStar System [Huber 2002]. System dynamics is particularly useful for gaining insight into difficult management situations in which best efforts to solve a problem actually make it worse. Real problematic situations in which system dynamics helps to create clarity include the following [Sterman 2000]:

- Efforts to build new roads to alleviate traffic congestion only result in increased congestion.
- Use of cheaper drugs pushes costs up, not down.
- Lowering the nicotine in cigarettes, supposedly to the benefit of smoker's health, only results in people smoking more cigarettes and taking longer, deeper drags to meet their nicotine needs.
- Levee and dam construction to control floods leads to more severe flooding by preventing the natural dissipation of excess water in flood plains.
- Applying more resources to incident response to handle a high workload takes resources from proactive management activities and increases the incident workload.

Here system dynamics targets problematic behavior associated with business operations in general and IT management in particular. Intuitive solutions to problems in this area often reduce the problem in the short term, but make it much worse in the long term. System dynamics is a valuable analysis tool for gaining insight into solutions that are effective over the long term and demonstrating their benefits.

A powerful tenet of system dynamics is that the dynamic complexity of problematic behavior is captured by the underlying feedback structure of that behavior. So we decompose the causal structure of the problematic behavior into its feedback loops to understand which loop is strongest (i.e., which loop's influence on behavior dominates all others) at particular points through time. We can then thoroughly understand and communicate the nature of the problematic behavior and the benefits of alternative mitigations.

System dynamics model boundaries are drawn so that all the enterprise elements necessary to generate and understand problematic behavior are contained within them. This approach encourages the inclusion of soft factors in the model, such as policy, procedural, administrative, or cultural factors along with hard, strictly technical factors. The exclusion of soft factors essentially treats their influence as negligible when in fact it is frequently significant. This endogenous, or internal viewpoint helps show the benefits of mitigations to the problematic behavior that are often overlooked by low performers, partly due to their narrow focus on technical solutions to resolve problems.

We rely on system dynamics as a tool to help test the effects of strategies for improving the performance of complex systems such as organizations. In some sense the simulation of the model will help predict the effects of these strategies, but the nature of the types of predictions that system dynamics facilitates may not be clear. Dennis Meadows offers a concise answer by categorizing outputs from models [Meadows 1974]:

1. absolute and precise predictions (Exactly when will the next release be delivered?).
2. conditional precise predictions (If the system has more defects than anticipated, then how much will it cost the organization to repair them?)
3. conditional imprecise projections of dynamic behavior modes (If I adopt a different development process, will my organization's performance be better than it would have been otherwise?)
4. summary and communication of current trends, relationships, or constraints that may influence the future behavior of the system (If the current trends in productivity and defect density continue, what effect will it have on the organization's ability to deliver system over the next five years?)
5. philosophical explorations of the logical consequences of a set of assumptions, without any necessary regard for the real-world accuracy or usefulness of those assumptions (How would the use of artificially intelligent software developers affect the organization's ability to develop and maintain robust systems over their life cycle?)

The model we develop, and system dynamics models in general, provide information of the third sort. Meadows explains further that "this level of knowledge is less satisfactory than a perfect, precise prediction would be, but it is still a significant advance over the level of understanding permitted by current mental models."

Notation

In graphic representations of the model we describe, signed arrows represent the system interactions, where the sign indicates the pair-wise influence of the variable at the source of the arrow on the variable at the target of the arrow:

- Informally, an arrow labeled with a "+" indicates that the value of the source and target variables move in the same direction.[26]
- Informally, an arrow labeled with a "-" indicates that the value of the source and target variables move in the opposite direction.[27]

[26] More formally, a positive (+) influence indicates that if the value of the source variable increases, then the value of the target variable increases above what it would otherwise have been, all other things being equal. And, if the value of the source variable decreases, then the value of the target variable decreases below what it would otherwise have been, all other things being equal.

[27] More formally, a negative (-) influence indicates that if the value of the source variable increases, then the value of the target variable decreases below what it would otherwise have been, all other things being equal. And, if the value of the source variable decreases, then the value of the target variable increases above what it would otherwise have been, all other things being equal

We can illustrate the above definitions using the influence diagram shown in Figure 17, which represents a very simple room heating system. A positive influence is indicated by the arrow from *rate of heat input* to *room temperature*. At a particular thermostat setting, as the rate of heat input increases (or decreases), then the temperature of the room increases (or decreases) above (or below) what it would have been. A negative influence is indicated by the arrow in the other direction. As the room temperature increases (or decreases), the rate of heat input decreases (or increases) below (or above) what it would have been, as would be expected by a room heating system.

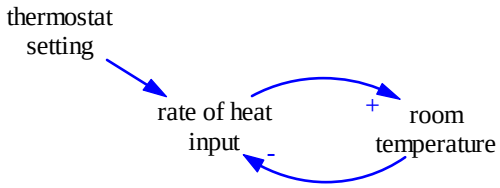


Figure 17: A Simple Feedback Loop

As mentioned previously, dynamically complex problems can often be best understood in terms of the feedback loops underlying those problems. There are two types of feedback loops: balancing and reinforcing. Balancing loops describe aspects of the system that oppose change, seeking to drive organizational variables to some goal state. Reinforcing loops describe system aspects that tend to drive variable values consistently upward or consistently downward. The polarity of a feedback loop is determined by "multiplying" the signs along the path of the loop. Balancing loops have negative polarity and reinforcing loops have positive polarity.

Figure 17 depicts a balancing loop that seeks to move the room temperature to the thermostat setting. This system is balancing as shown by the odd number of negative signs along its path. The goal state is a room temperature equal to the thermostat setting. In general, balancing loops describe aspects that oppose change, and usually involve self-regulation through adaptation to external influences.

Figure 18 shows a more interesting example in the domain of project management. Figure 18 (a) depicts one approach an organization may adopt to try to put a project that is behind schedule back on track: having its employees work overtime. The closed form in Figure 18 (b) shows the corresponding balancing feedback loop that characterizes the goal of the approach as moving the project to the state of being on schedule.

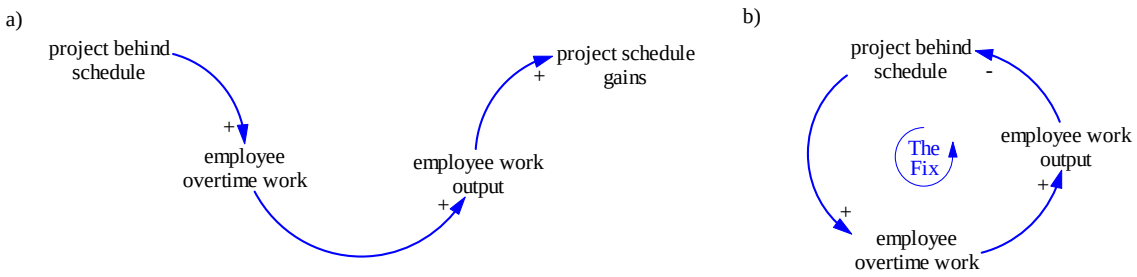


Figure 18: (a) Project Management—Desire to Use Overtime to Correct Schedule; (b) Closed-Loop Representation Showing (Balancing) Feedback to Improve Progress

Figure 19 shows that the project-management behavior described above is subject to a reinforcing feedback loop in which overtime in the long term leads to employee burnout, lower quality of work, and the need to rework defective artifacts. The longer this goes on the further the project gets behind schedule because of the increasing amount of rework. This combines with the previous balancing feedback loop, where the balancing loop dominates in the near term with the reinforcing loop taking over with increasing amounts of employee overtime and burnout. This type of thinking about the feedback structure of systems and about which feedback loops dominate at different periods in time is characteristic of system dynamics modeling and analysis.

The reinforcing loop is shown mostly in red but it shares part of the influence path of the blue balancing loop from *project behind schedule* to *employee overtime work*. The reinforcing nature of the feedback loop is evident from the even number of negative signs along its path.[28] Reinforcing loops may help explain explosive growth or implosive collapse of a system.

[28] Feedback loops that have no negative signs along the influence path have positive polarity and thus are reinforcing loops.

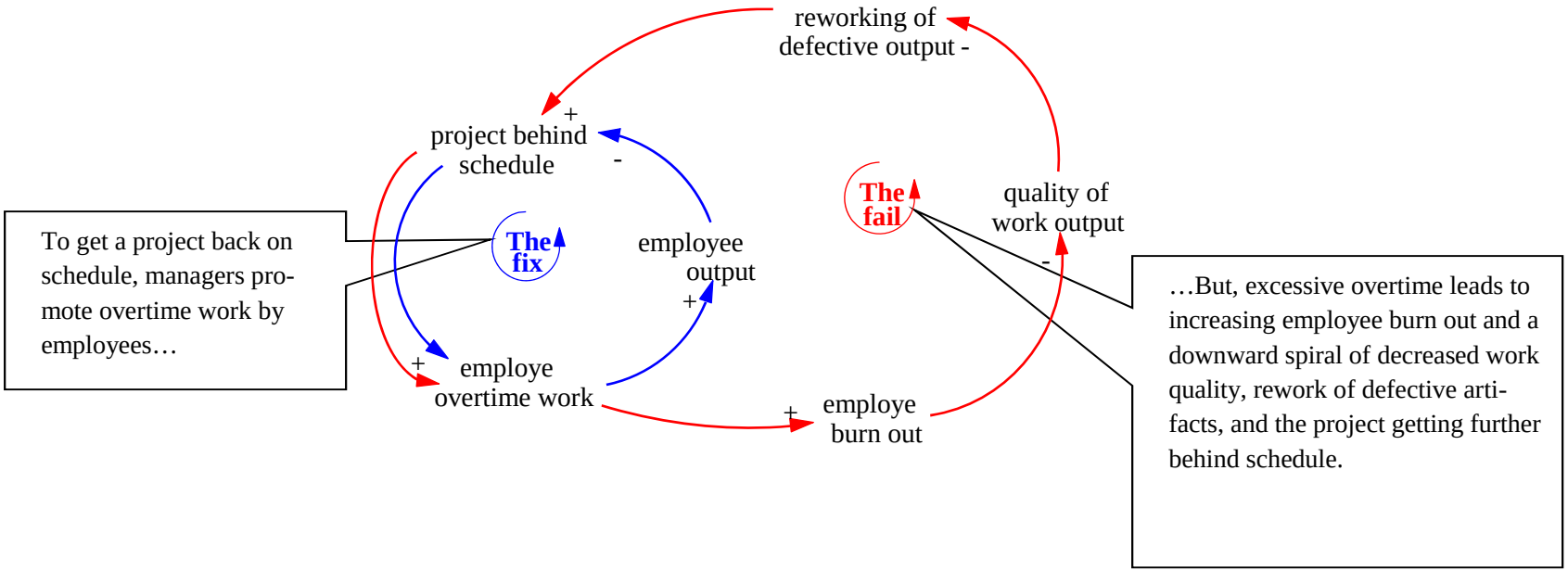


Figure 19: Unintended Burnout Due to Overtime

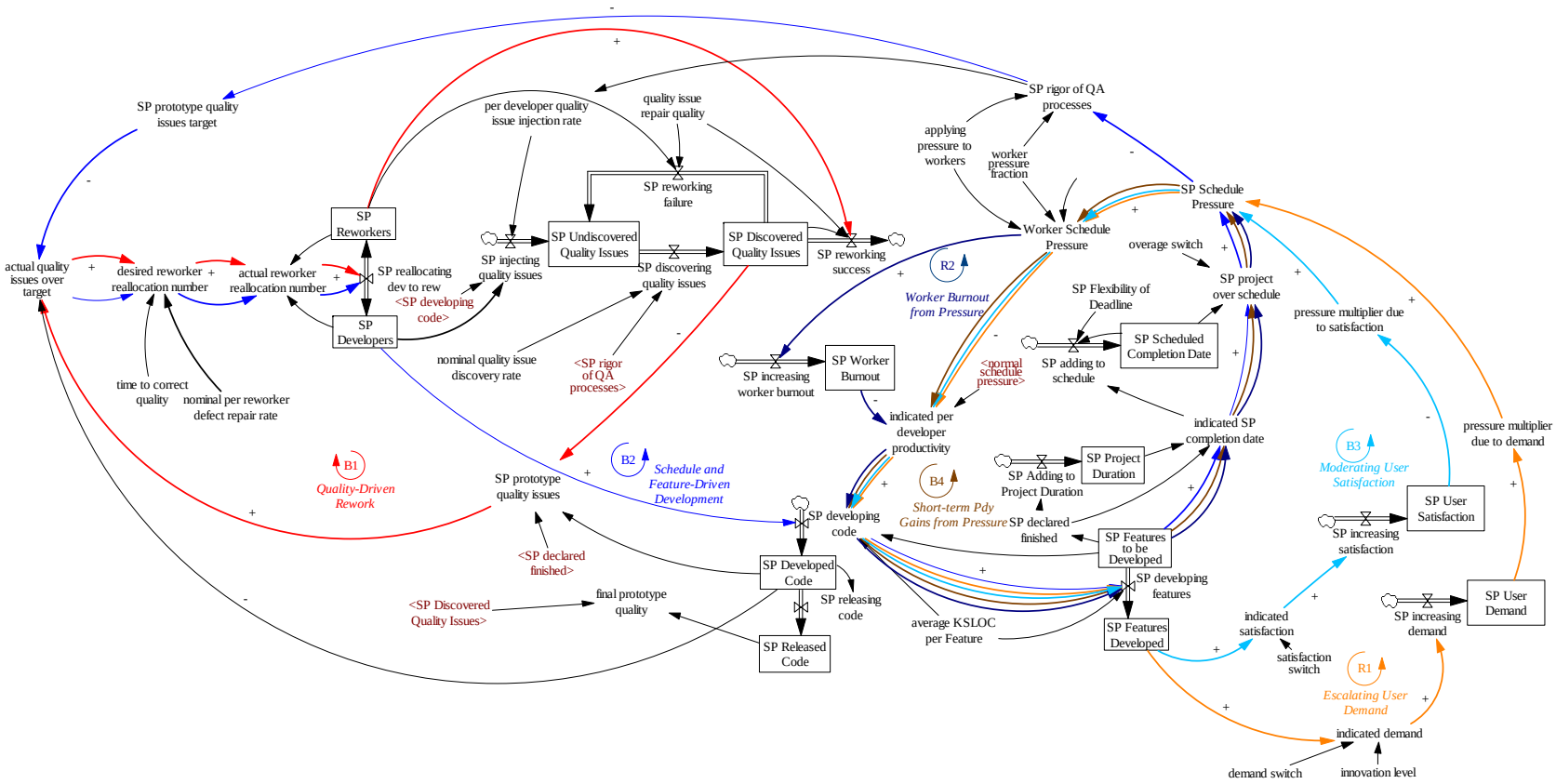
7.6 Recommended Resources for System Dynamics Modeling

Some essential resources for working with system dynamics modeling include the following:

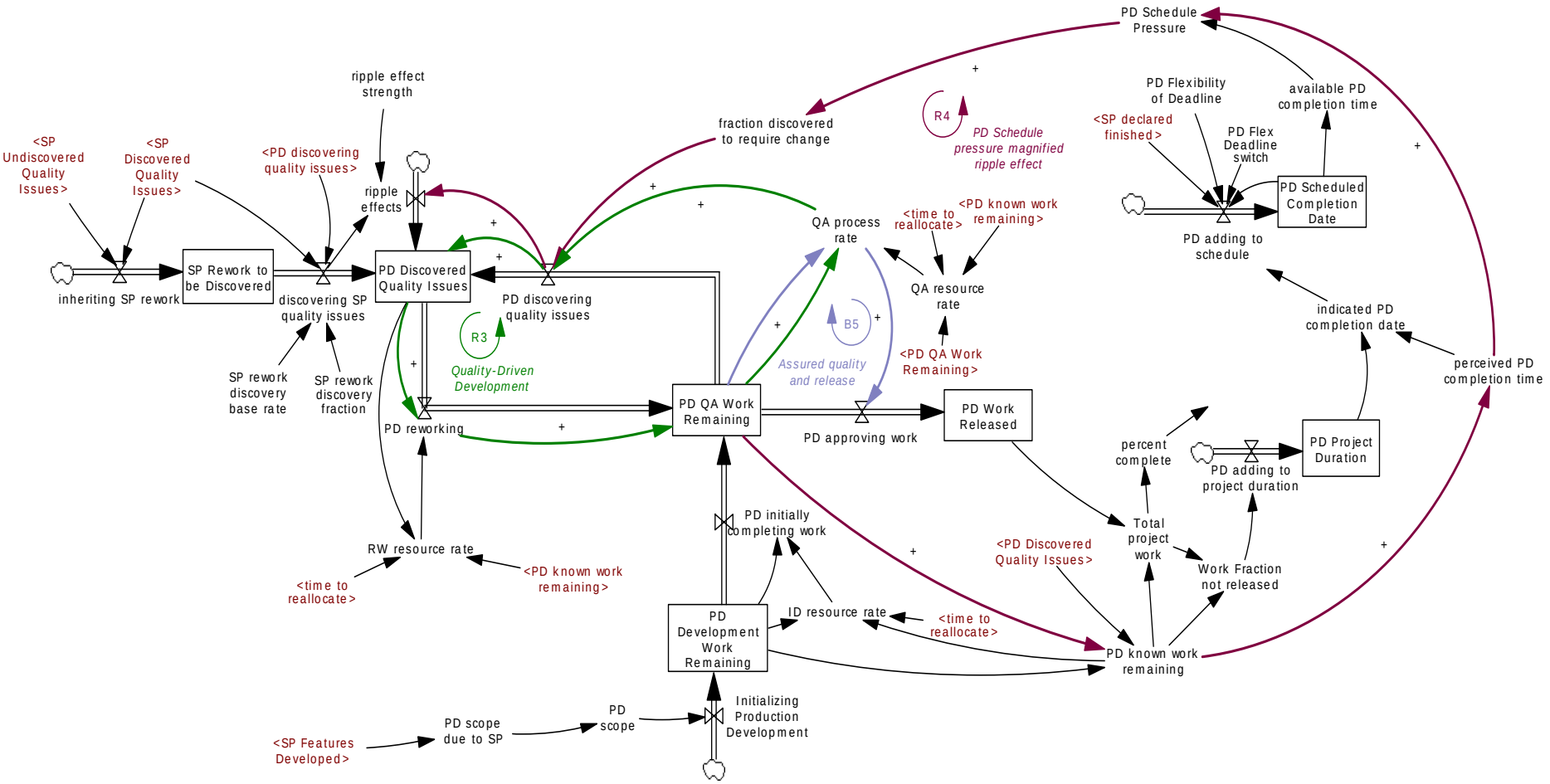
- The Kirkwood system dynamics tutorial (http://www.public.asu.edu/~kirkwood/sysdyn/SDIntro/SDIntro.htm)
- Donella Meadows' book *Thinking in Systems: A Primer* (Chelsea Green Publishing, 2008)

- John D. Sterman's book, *Business Dynamics: Systems Thinking and Modeling for a Complex World* (McGraw-Hill 2000)
- The Ventana Systems website (http://www.vensim.com), creators of the Vensim system dynamics modeling tool.
- The isee Systems website, creators of the Stella and iThink products (http://www.iseesystems.com)
- The PowerSim website (http://www.powersim.com)

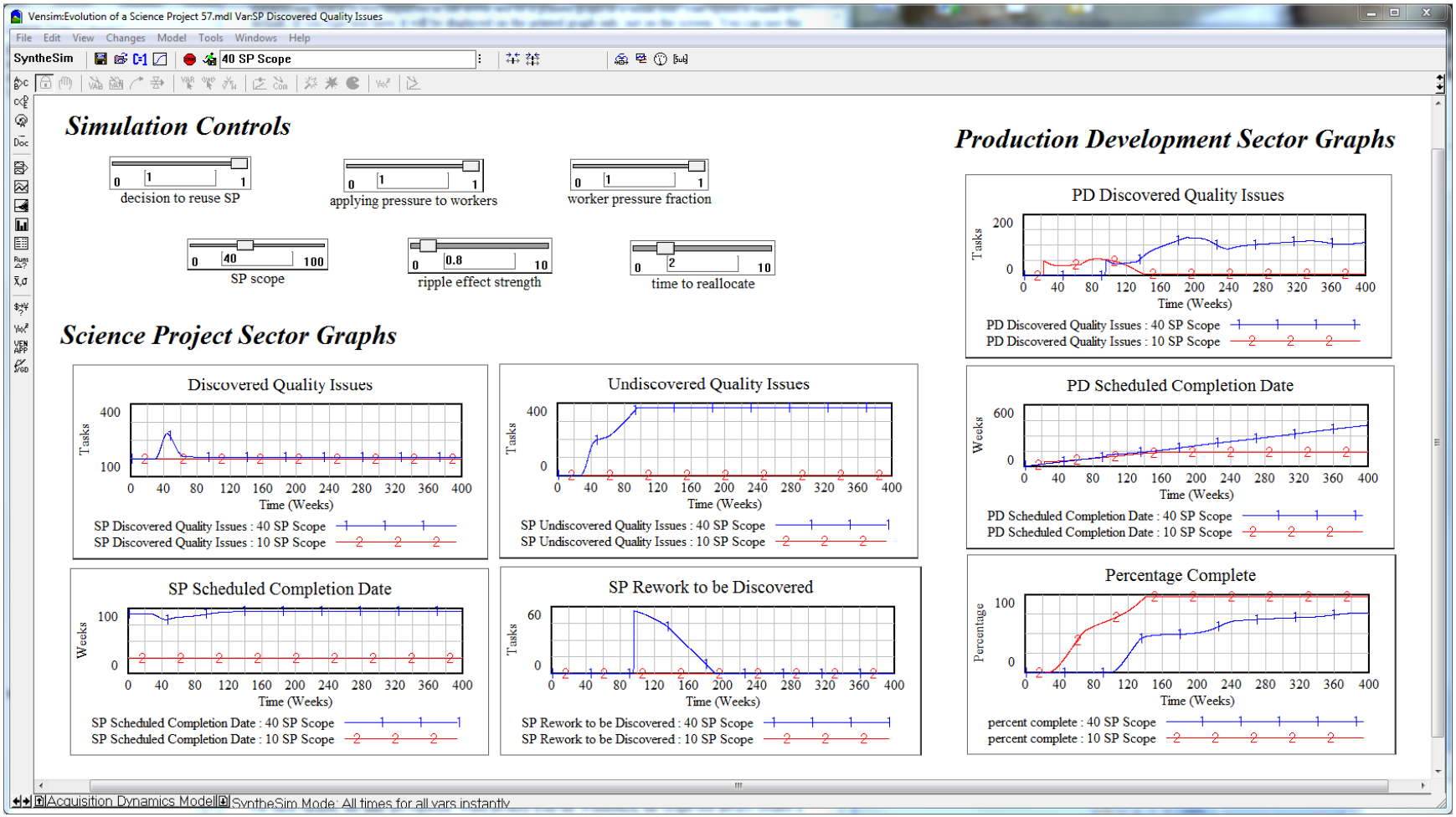
Appendix C: System Dynamics Simulation Model: Science Project Sector



Appendix D: System Dynamics Simulation Model: Production Development Sector



Appendix E: Interface for Interacting with the Model

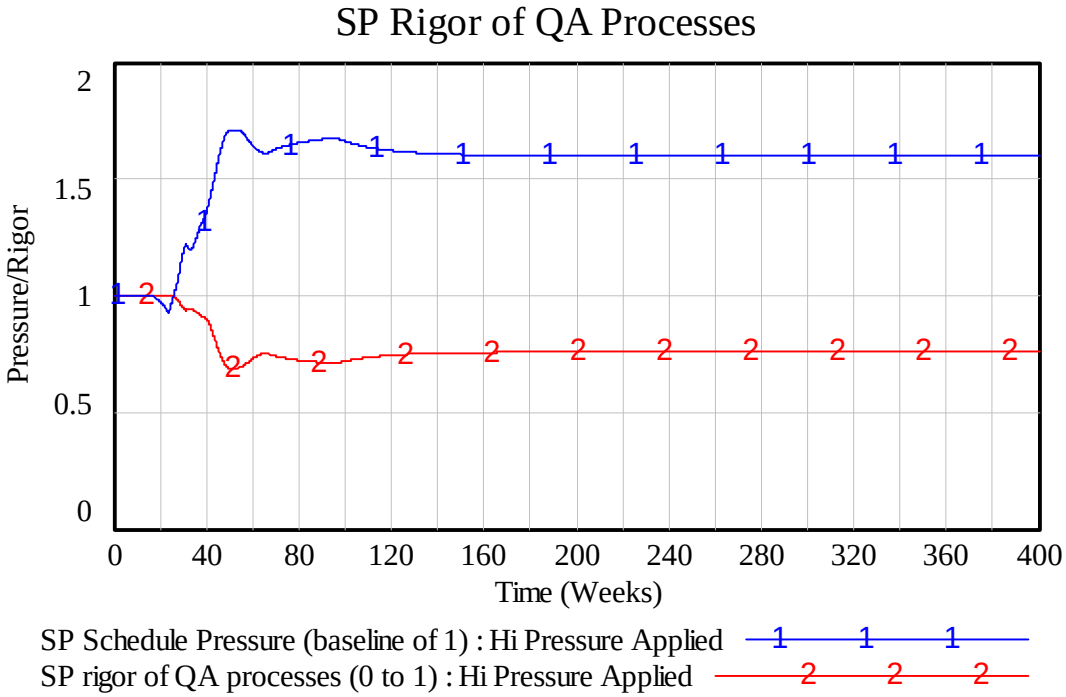


Note that in this user interface, there are two user controls provided to change the amount of schedule pressure that is being applied to developers: applying pressure to workers, and worker pressure fraction. Although it's not discernible in the screen capture, the control for applying pressure to workers is binary, either allowing the pressure, or not. The control for worker pressure fraction is a real number, and varies continuously between 0 and 1 to define the precise level of that pressure—but is only active if applying pressure to workers is also set to 1 (i.e., "true").

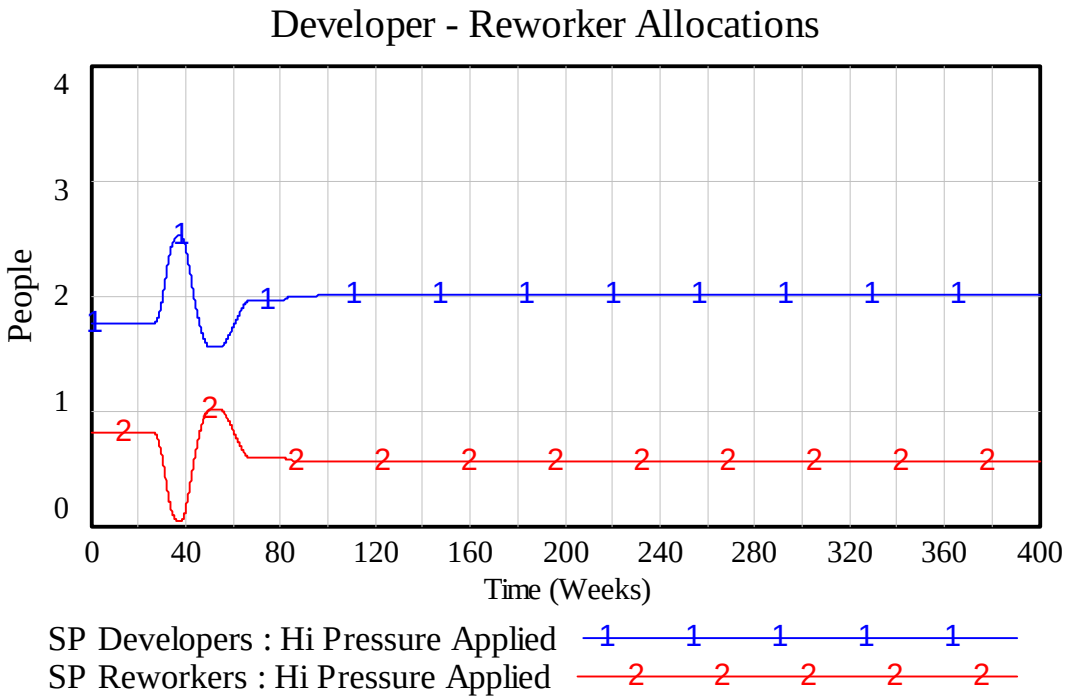
Appendix F: Simulation – Science Project Dynamics

The graphs in this appendix were generated using the Vensim modeling and simulation tool. These are all behavior-over-time graphs and, as such, the X-axis for these graphs is specified in weeks (400 weeks is the duration of the simulation). Each simulation run is specified as individual graphs distinguished both in color and with a number label, as specified in the legend below the graph. Each simulation run varies a key parameter—all other variables remain the same for the runs. The title at the top of the graph indicates the variable being graphed. The label on the Y-axis indicates the measurement units for the variable that is being graphed. To get a more complete explanation of the dynamic and its relationship to the model, please read Section 4 prior to reviewing the graphs in this appendix.

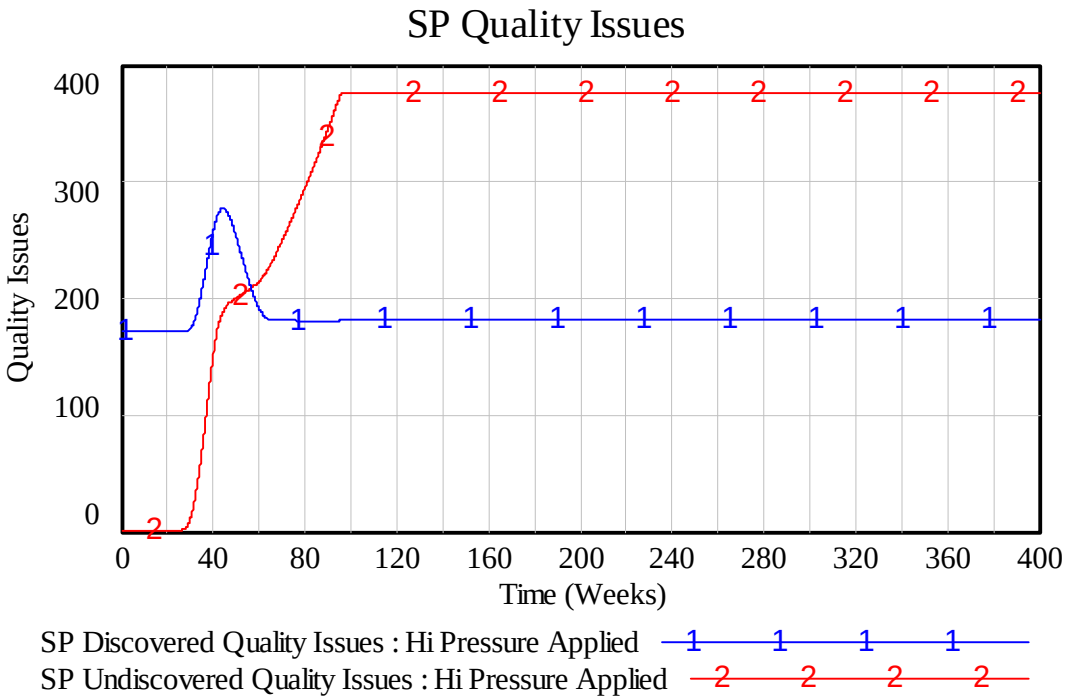
The graph below shows the drop in rigor of QA processes during the Science Project development when schedule pressure increases. Schedule pressure is measured with a baseline of 1, whereas the rigor of QA processes is measured on a scale of 0 to 1. Remember that the science project ends relatively early in the timeline, so the variables level off after week 120.



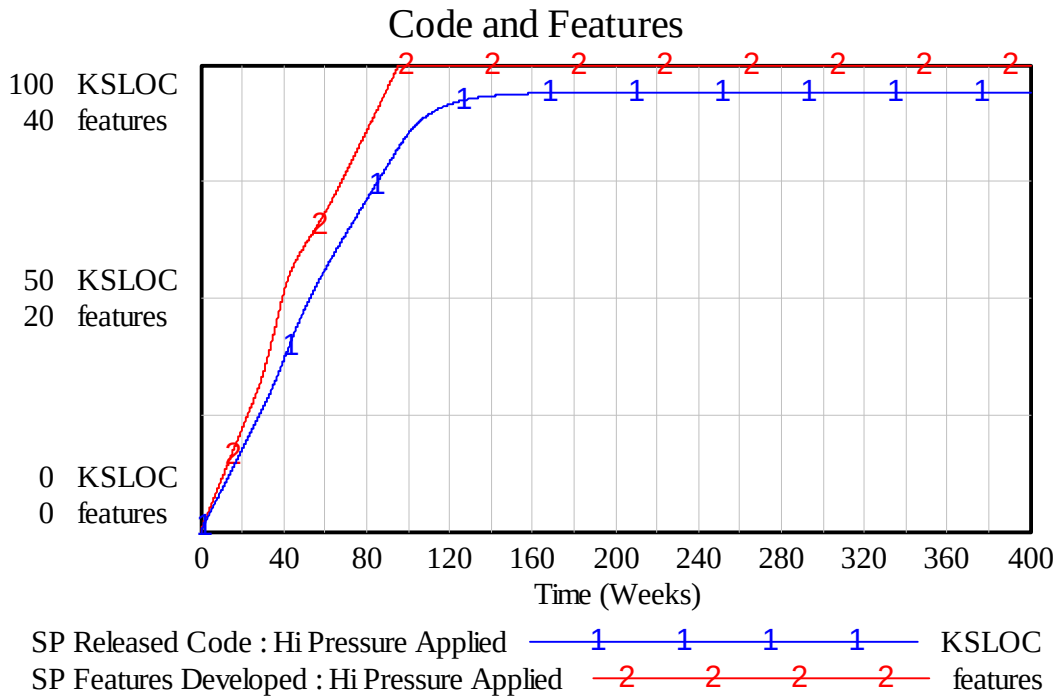
The next graph shows the movement of Science Project staff resources (people) between development and rework that results when a high level of pressure is applied to the staff in the face of schedule pressure affecting the project. The mirror image nature of the staff changes reflects the fact that a fixed pool of people is assumed.



The graph below shows that with high application of schedule pressure to workers comes a tendency to leave high levels of undiscovered quality issues at the end of the science project development.

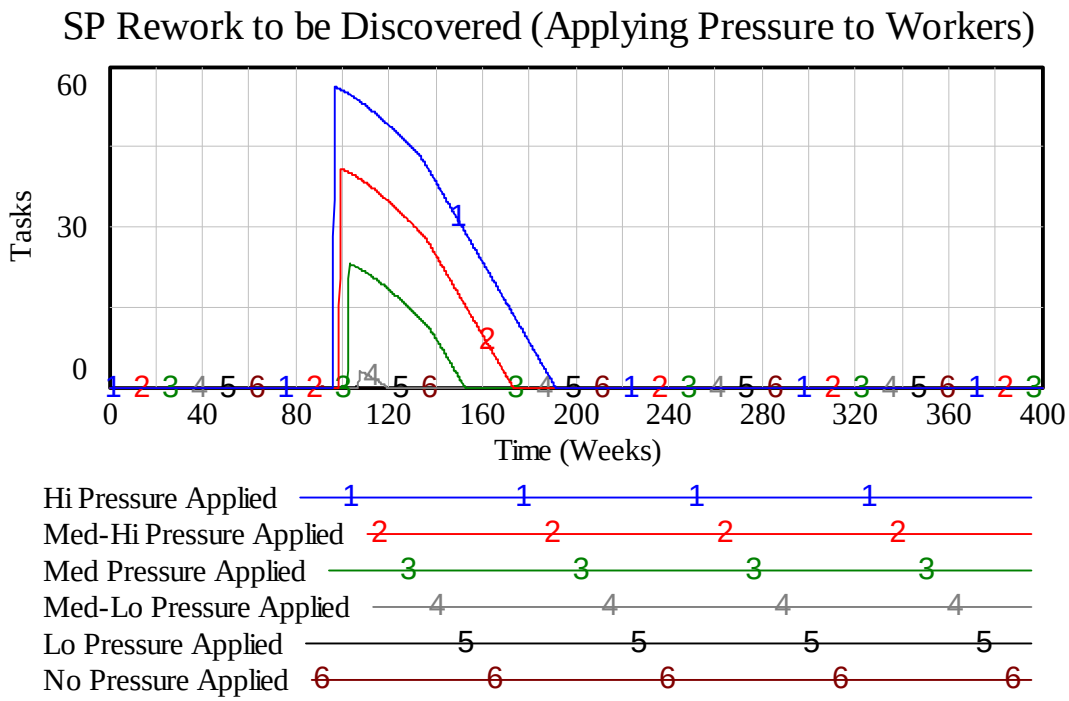


This graph shows the steady rise of Science Project released code and features developed as a result of the policy to favor development in the face of schedule pressure.

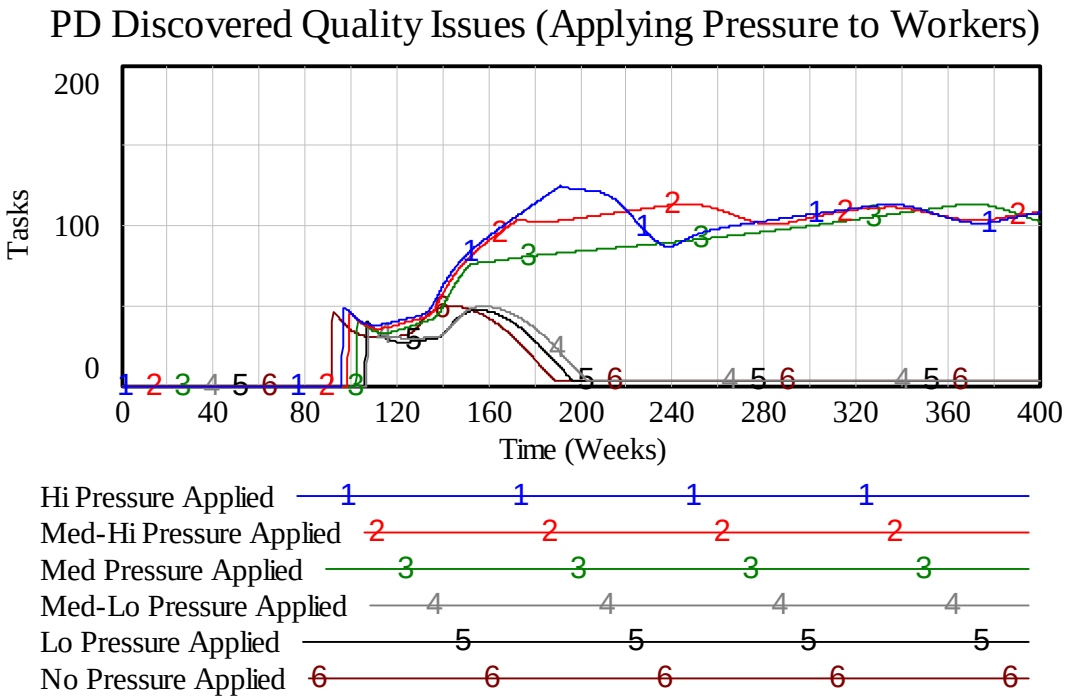


Appendix G: Simulation - Applying Pressure to Workers

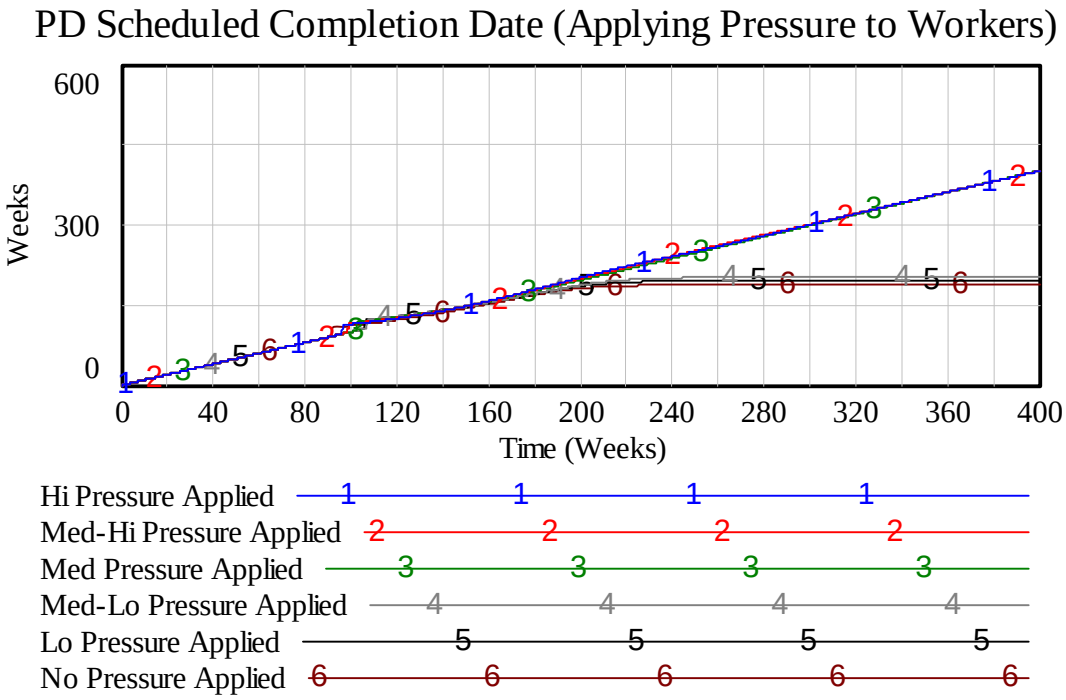
This appendix shows the simulation graphs for key variables that are relevant during the Production Development. As such, most of these graphs only start to display interesting behavior after the conclusion of the Science Project at approximately week 100. The key variables are graphed with varying levels of pressure applied to workers on the science project from none to high. The graph below shows the SP rework to be discovered for these various levels of pressure application. High pressure results in the greatest levels of undiscovered SP rework that are then inherited by the production development.



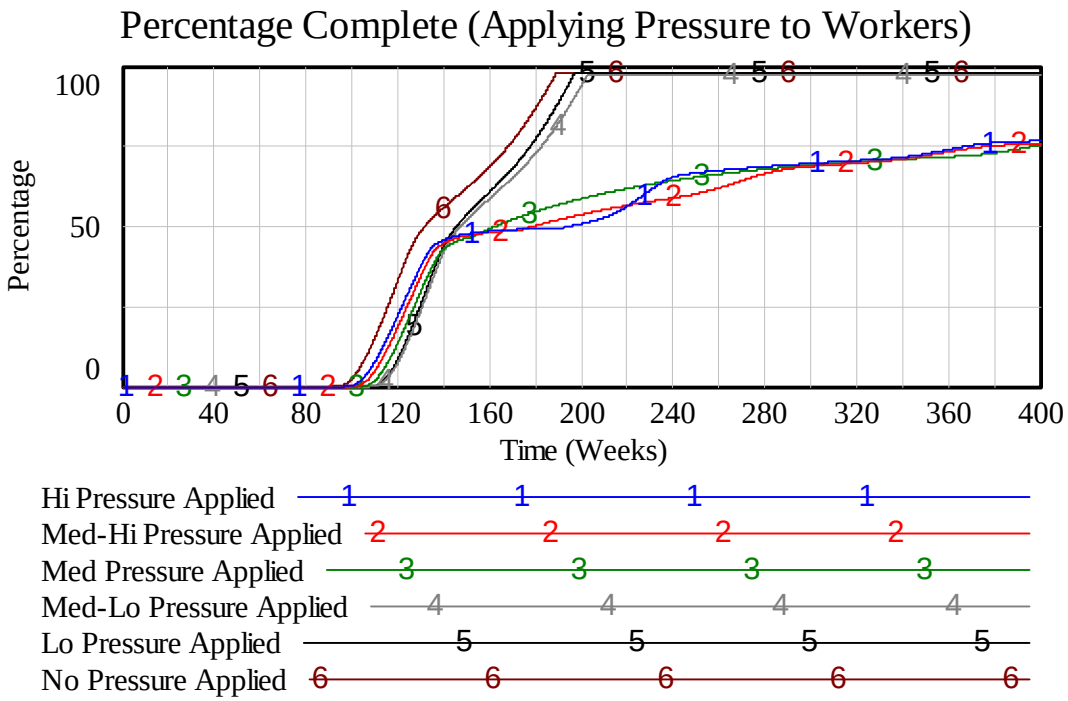
The tipping point dynamic is seen in the graph below. During times of high schedule pressure, applying full pressure to the workers might notionally mean that the workers' jobs would be in jeopardy if they did not put forth their full effort, whereas medium pressure might mean that their performance evaluations would be significantly impacted. The graph shows that applying pressure at the Medium level and above pushes the discovery (and generation) of quality issues over the tipping point towards more and more rework. Applying pressure at a Medium-Low level and below results in completion of the project.



The graph below shows the estimated number of weeks to completion of the production development effort for various levels of pressure applied to workers. The lower levels of pressure lead to completion around week 180, whereas higher levels of pressure lead to a production development that does not terminate (i.e., the estimated completion date continues to rise as the effort goes on).



The next graph shows that the only place that the 90% Syndrome is seen is where the project has gone past the tipping point, i.e., where greater than moderate pressure is applied to workers to meet schedule demands in simulation runs 4, 5 and 6.[29] It is significant that the simulation's behavior in seeing forward progress plateauing well before project completion is consistent with the predictions of the 90% Syndrome. The fact that the simulation plateaus closer to the 80% level may be due to the fact that the model takes into account all rework remaining in its calculations.

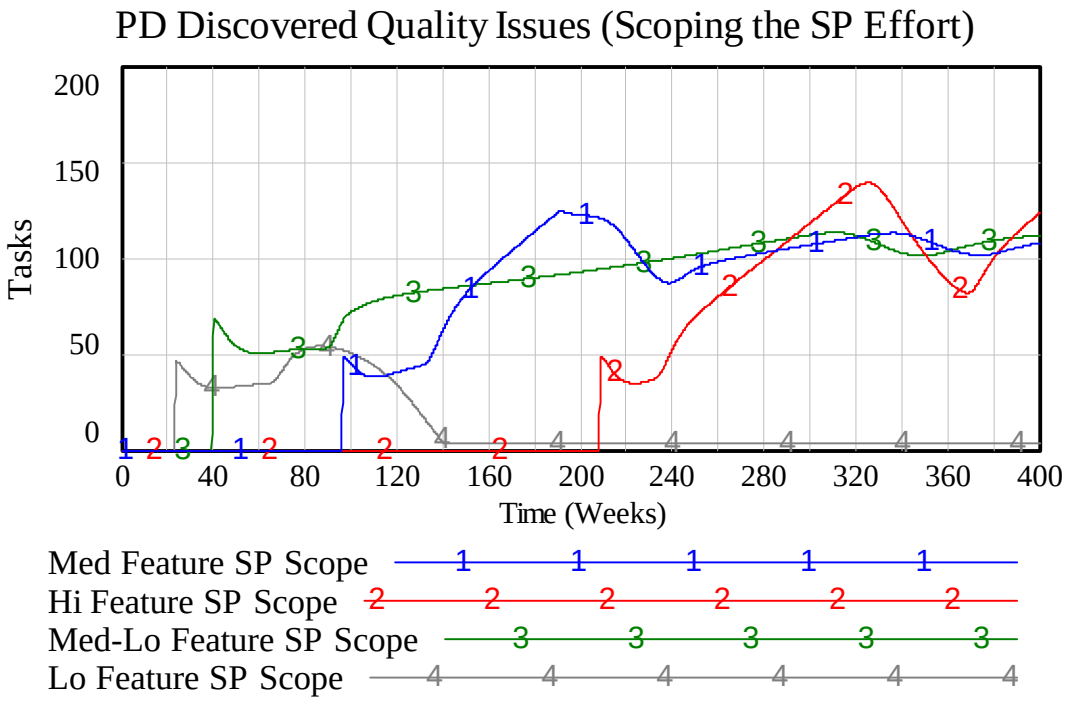


[29] The simulation runs for Percentage Complete at the lower pressure levels (runs 4, 5, and 6) shown in the figure do not actually show 100% completion in the second half of the simulation. The project does terminate, but some minimal level of rework is allowed to remain even at project conclusion, i.e., not all loose ends are completely tied up.

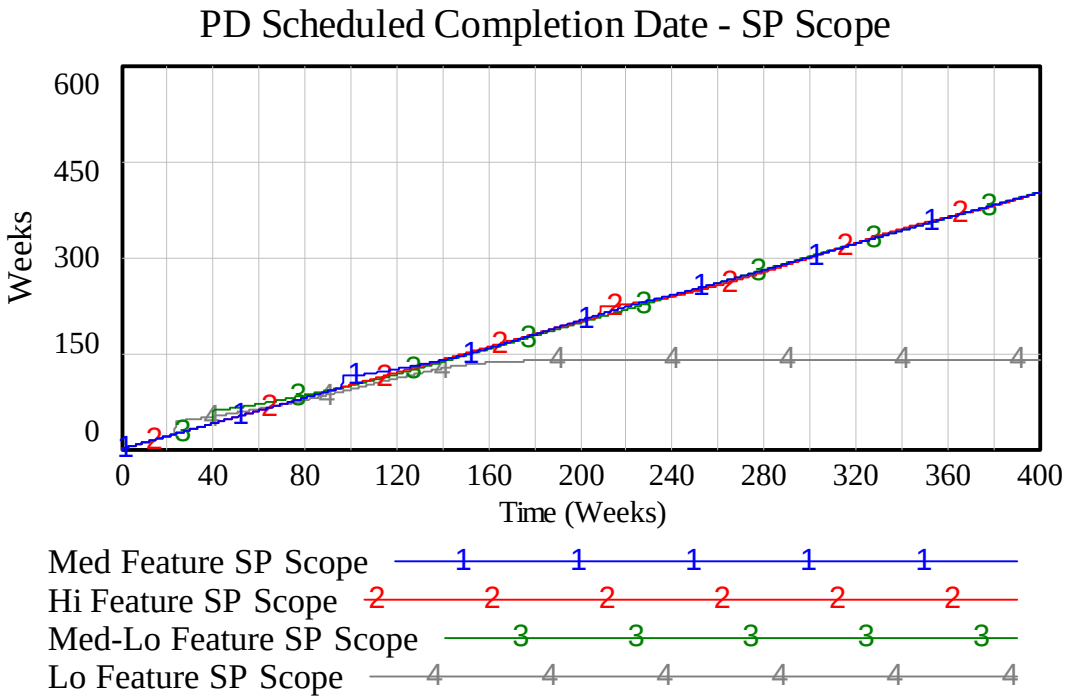
Appendix H: Simulation – Scoping the Science Project Effort

This appendix shows the simulation graphs for key variables that are relevant during the Production Development. The key variables are graphed with varying qualitative levels of scoping of the science project from low to high. All other variables remain the same for the four simulation runs. We assume that a high level of pressure is applied to workers in the face of schedule pressure on the project.

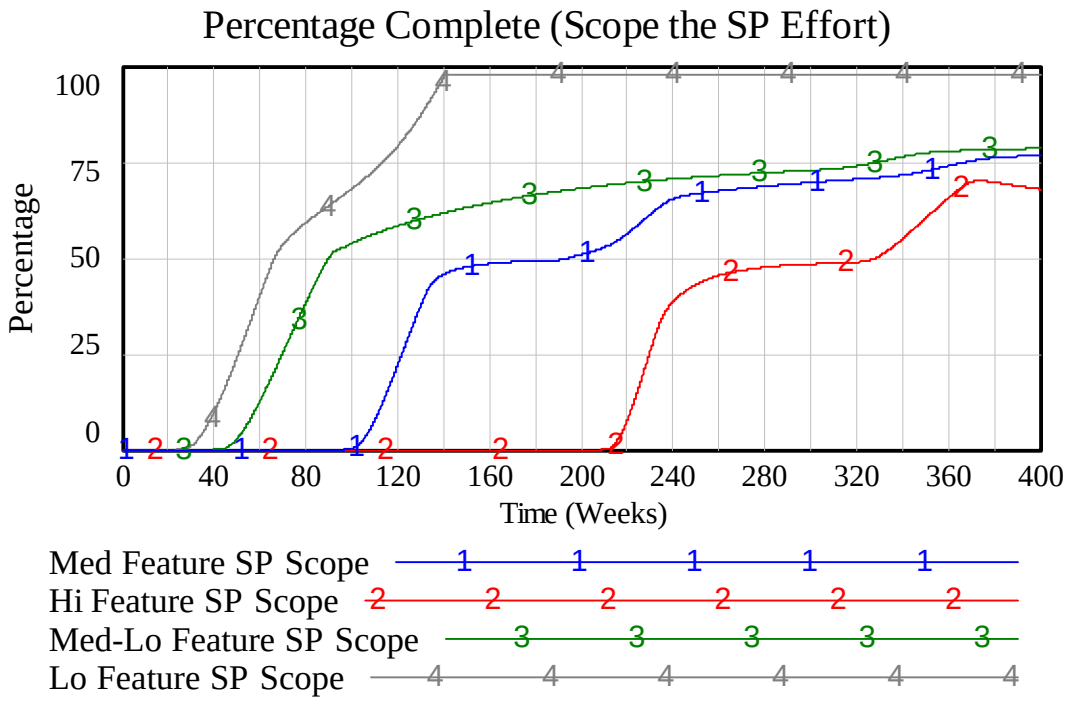
The graph below shows the accumulation of PD Discovered Quality Issues when the scope of the SP effort is varied. To simplify the analysis we assume that all schedule pressure is applied to the workers as the science project develops the prototype, so the simulation goes past the tipping point in the case of a medium-feature scope. As is apparent from the figure, going to a high-feature scope (in simulation run 2) causes a later start of the production development, again surpasses the tipping point, and results in higher amplitude oscillations. Reducing the scope to medium-low (in run 3) is still beyond the tipping point, but results in a significantly damped oscillation. Finally, reducing the scope of the development to a low level (simulation run 4) results in successful completion of the production development at approximately week 140, without the project passing the tipping point.



The next graph shows that a small scope for the science project effort leads to completion of the production development, whereas larger scoping leads to a non-terminating production development effort.

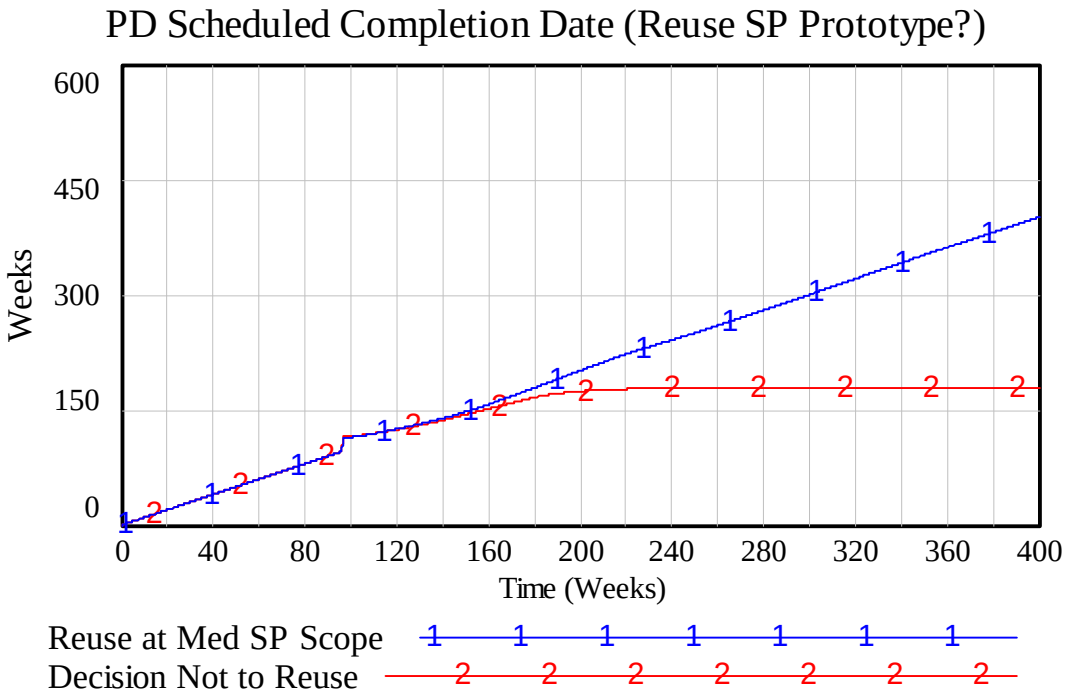


The last graph below also shows that anything larger than a small scoped SP effort leads to the 90% Syndrome, or even worse outcomes.

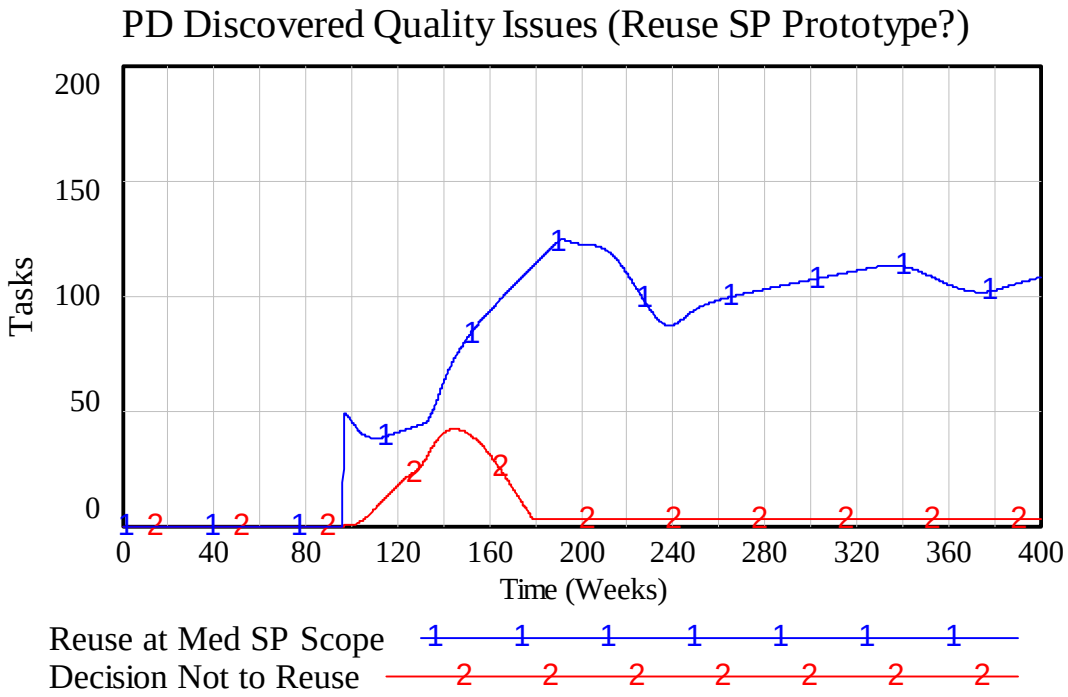


Appendix I: Simulation – Decision to Reuse the Prototype

This appendix shows the simulation graphs for key variables that are relevant during the Production Development. The key variables are graphed for two cases: one in which the SP prototype is used in the production development, and one in which it is not. All other variables remain the same for the two runs. We assume that a high level of pressure is applied to workers in the face of schedule pressure. The first graph shows that the production development completes when there is no reuse of the science project prototype. Reuse of the prototype that was produced with high worker pressure leads to non-terminating production development, as has been shown previously.



The next graph shows that the reason for the non-terminating production development phase when reusing the science project prototype is the large amount of quality issues that accrue.


PD Discovered Quality Issues (Reuse SP Prototype?)
Tasks
200
150
100
50
0
0
40
80
120
160
200
240
280
320
360
400
Time (Weeks)
Reuse at Med SP Scope
Decision Not to Reuse

Appendix J: Potential Enhancements to the Model

The following list is a set of potential enhancements that could be made to the system dynamics model of "The Evolution of a Science Project."

- The model does not yet address the fact that continuous development under intense schedule pressure is making the prototype code increasingly complex and is undermining the code quality, thus making it increasingly difficult to maintain.
- The model does not yet represent the way that maintainability (and potentially system performance) degrades as the low quality of the original architecture is unable to support the type and amount of functionality that is now being added.
- The model does not yet address the fact that as continuing schedule pressure leads to stress and burn-out, some percentage of developers (and managers) will eventually leave the effort. This will require the manager to add more staff to the development team, with the learning curve effects of bringing on new hires with less experience, and making them productive. The learning curve would be affected by the code quality and the documentation quality (and in turn would likely affect those aspects as well), which in the case of a science project are both likely to be poor.
- The model does not yet explicitly address the lack of software acquisition expertise in the original project (i.e., that would otherwise have allowed them to recognize the problems with such approaches as building a production system on top of a throwaway prototype), and thus both the technical and programmatic aspects of the transition from the original project to a full acquisition program. The difference in training and background between the operational people managing the science project development, and the acquisition staff managing the program, will likely result in significantly different decisions being made. This transition may also include changing contractors—by keeping the original contractor (if there were one) the program keeps the technical knowledge of the system, but doesn't have the formal acquisition expertise, whereas by changing the contractor the program gains acquisition expertise, but loses the hard-won technical knowledge of the system.
- The model does not yet address the incentive that is at work on the original prototype developer to continue to develop on top of the original prototype code base, as a complete redesign would be an opportunity for the government to change to a different production contractor, in which case the original contractor would lose out on doing the work.
- The model does not yet address system documentation (including code commenting) quality as a significant aspect of system quality, as it speeds the process of going up the learning curve for new hires, likely improving the speed and quality of rework.
- The model does not yet address the quality of the user documentation, and the potential connection of that to user demand.

- The model does not yet address the distinction between quality issues in the design (i.e., requirements and architecture), and quality issues in the implementation (i.e., design and coding), and the well-documented differences in the cost of fixing the resultant issue(s).
- The model does not yet address the developer effort that would be siphoned off to support the deployed prototype in the field, which would likely sap project development resources, and further slow progress.
- The model does not yet address the fact that even a prototype that has been thrown away helps the subsequent production development effort because it provides clarification of the requirements, as well as a better understanding of the overall problem, and of what some of the most significant challenges for the system and the system architecture might be.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2012	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
The Evolution of a Science Project: A Preliminary System Dynamics Model of a Recurring Software-Reliant Acquisition Behavior	FA8721-05-C-0003

6. AUTHOR(S)

William E. Novak

Andrew P. Moore

Christopher Alberts

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2012-TR-001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ESC/CAA 20 Schilling Circle, Building 1305, 3rd Floor Hanscom AFB, MA 01731-2125	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Analysis work by the SEI on data collected from more than 100 Independent Technical Assessments (ITAs) of software-reliant acquisition programs has produced insights into some of the most common ways that programs encounter difficulties.

This report describes work done at the SEI that is based on these insights, and intends to mitigate the effects of both misaligned acquisition program organizational incentives, and adverse software-reliant acquisition structural dynamics, by improving acquisition staff decision-making. The research presented here uses a preliminary system dynamics model to analyze a specific adverse acquisition dynamic concerning the poorly controlled evolution of small prototype efforts into full-scale systems that is called "The Evolution of a Science Project."

The report provides a narrative from an actual acquisition program that exemplifies the dynamic, qualitatively describes its key aspects, and presents some of the most relevant prior research done on one of those aspects, project rework. The system dynamics model of the behavior is described in detail, along with the process by which it was developed, and the results of simulations run using the model. The report concludes with a set of lessons learned about system dynamics modeling, as well as potential future research directions.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Acquisition, system dynamics, modeling	90

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102